FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation

Continuous User Understanding in Software Evolution

Jan Ole Johanßen

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Forschungs- und Lehreinheit 1
Angewandte Softwaretechnik

Continuous User Understanding in Software Evolution

Jan Ole Johanßen

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Florian Matthes
Prüfende/-r der Dissertation:	1. Prof. Dr. Bernd Brügge
	2. Prof. Dr. Barbara Paech

Die Dissertation wurde am 05.09.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 31.10.2019 angenommen.

Abstract

Explicit user feedback represents an important source of knowledge for software developers. However, there are deficiencies in current practices of user feedback capture and utilization as they do not address the users' tacit knowledge.

Our research goal is to establish continuous user understanding so that developers can extract tacit knowledge of users for utilization in software evolution. We build on the foundations of continuous software engineering (CSE), which comprises the frequent and rapid delivery of software increments to obtain instant and diverse user feedback. We argue that in particular implicit user feedback serves as a suitable means that contributes to the extraction of users' tacit knowledge.

We conducted a semi-structured interview study with 24 practitioners from 17 companies to explore how practitioners apply CSE during software evolution and how the availability of user feedback supports CSE. We identified five recommendations for continuous user feedback capture and utilization of requirements.

The main contribution of this dissertation is CuuSE, a framework for continuous user understanding in software evolution. It collects user feedback through so-called Cuu kits, which unobtrusively capture and automatically process user feedback. CuuSE relies on feature crumbs to map user feedback to a feature under development. Another contribution is the development of a reference platform for CuuSE that visualizes user feedback. The platform's feasibility and extensibility were demonstrated by implementing four Cuu kits, namely EmotionKit, ThinkingAloudKit, BehaviorKit, and PersonaKit. CuuSE also includes a workflow that guides developers in applying the framework to extract users' tacit knowledge.

CuuSE has been validated in three cycles. In a laboratory experiment with 12 participants, we validated the applicability and feasibility of EmotionKit. The results indicate that emotional responses derived from facial expressions enable the detection of usability problems. In a quasi-experiment, we introduced the Cuu syllabus as an instrument to disseminate the framework and confirmed its applicability with 9 developers. With the help of a survey, we investigated developers' perceived usefulness and ease of use, as well as their intend to use CuuSE. The majority of 10 developers exhibited a positive attitude toward the CuuSE framework and provided helpful suggestions for future improvements.

Zusammenfassung

Nutzerreaktionen sind eine wichtige Wissensquelle für die Anwendungsentwicklung. Aktuelle Praktiken zur Erfassung und Verwendung von Nutzerreaktionen berücksichtigen allerdings nicht die unausgesprochenen Eindrücke von Nutzenden.

Das Forschungsziel dieser Dissertation ist ein kontinuierliches Verständnis von Nutzenden, sodass Entwickelnde unausgesprochene Eindrücke verstehen und für die Verbesserung ihrer Anwendung verwenden können. Unsere Hypothese ist, dass sich insbesondere implizite Nutzerreaktionen eignen, um unausgesprochene Eindrücke von Nutzenden extrahieren zu können. Um eine Vielzahl an Nutzerreaktionen einzuholen, verwenden wir einen kontinuierlichen Anwendungsentwicklungsprozess, der auf häufigen und schnellen Entwicklungszyklen basiert.

Im Rahmen einer Interviewstudie mit 24 Praktizierenden aus 17 Unternehmen haben wir zunächst untersucht, wie diese kontinuierlich Anwendungen entwickeln. Daraus haben wir fünf Empfehlungen für die kontinuierliche Erfassung und Verwendung von Nutzerreaktionen identifiziert.

CuuSE ist ein Framework für das kontinuierliche Verständnis von Nutzerreaktionen. Es umfasst einen Arbeitsablauf, der Entwickelnde bei dem Verständnis von Nutzerreaktionen unterstützt. Es sammelt Nutzungswissen mithilfe von Komponenten für die Erfassung und Verarbeitung von Nutzerreaktionen, sogenannten Cuu Kits. CuuSE baut auf Feature Crumbs auf, die eine Verbindung zwischen Nutzerreaktionen und Features herstellen. Die Machbarkeit und Erweiterbarkeit von CuuSE wurden durch die Entwicklung einer Referenzimplementierung und den Cuu Kits EmotionKit, ThinkingAloudKit, BehaviorKit, und PersonaKit demonstriert.

Zur Validierung von CuuSE wurden drei empirische Zyklen durchgeführt. In einem Experiment mit 12 Teilnehmenden wurde unter Laborbedingungen die Verwendbarkeit von EmotionKit untersucht. Die Resultate zeigen, dass emotionale Reaktionen aus den Gesichtszügen von Nutzern abgeleitet und für Rückschlüsse auf die Nutzerfreundlichkeit herangezogen werden können. Zur Verbreitung und Lehre von CuuSE wurde ein Syllabus entwickelt. In einem Quasi-Experiment mit 9 Entwickelnden wurde dessen Anwendbarkeit bestätigt. Eine Umfrage mit 10 Entwickelnden bestätigte die Nützlichkeit und Nutzerfreundlichkeit von CuuSE, sowie die Bereitschaft, CuuSE auch in zukünftigen Projekten zu verwenden.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Bernd Bruegge for the opportunity to join the Chair for Applied Software Engineering and to conduct research. I am thankful for his endless enthusiasm, countless new ideas that emerged from inspiring discussions, his trust in my work, his constant feedback, as well as his openness and willingness to share his experiences. I am grateful for the knowledge that I gained from Professor Barbara Paech and thank her for the precious and helpful feedback that she provided over the last years. The research collaboration with the Chair for Software Engineering at Heidelberg University is truly unique: I thank Anja Kleebaum, who became an indispensable colleague to discuss any aspect of our project and research work, which is reflected in the accomplishment of this dissertation. I thank the German Research Foundation DFG (Deutsche Forschungsgemeinschaft) for the support of this dissertation with the CURES project as part of the SPP1593 priority program.

I am grateful to be part of the LS1 family, a chair full of trust and cheerfulness. I thank all my colleagues, in particular Monika Markl, Helma Schneider, and Uta Weber for their support with any kind of organizational matter; Matthias Linhuber, Florian Angermeier, and Florian Bodlée for their help as system administrators and student assistants; Sebastian Jonas for his trust and support as a mentor; Stephan Krusche for his scientific guidance and the discussions about soccer; Lukas Alperowitz and Nadine von Frankenberg und Ludwigsdorff for not only sharing an office, but also ideas and laughs with me; Rana Alkadhi for the great collaboration and her scientific advises; Dominic Henze and Dora Dzvonyar for their unlimited supply with positive energy; Andreas Seitz, Constantin Scheuermann, and Marian Avezum for the enjoyable chats during coffee breaks and the afternoon walks; as well as Jan Philip Bernius and Lara Marie Reimer for their excellent work as research assistants.

I thank all students that participated in my research projects, in particular Thomas Günzel, Florian Schaule, Michael Fröhlich, Jan Philip Bernius, Lara Marie Reimer, Florian Fittschen, Nicolas Vorwerg, and Nityananda Zbil, as well as all participants of the interviews, the laboratory experiment, the quasi-experiment, and the survey.

I thank my friends for their continuous support, even when this meant to skip one or another board game night. Finally, I am deeply grateful for the endless and unconditional support of my family that laid the foundation for this dissertation.

To My Family.

Contents

Conventions

We rely on American English throughout this dissertation; the only exception are direct quotes. We apply the `typewriter font` to refer to code fragments or names, attributes, and methods of models. Text in *italic font* refers to labels, headlines, or names that are part of a figure or table; it can also indicate the emphasis of a notable word. We use *"italic font in quotes"* to denote direct citations of authors. We abbreviate terms that are frequently mentioned using SMALL CAPS and introduce the full descriptions when it the term is used for the first time in the text.

To highlight important extracts of text, such as the definition of a goal or a research question, we use boxes of the following style:

> **Title:** A short description.

To provide more visual guidance in case hierarchies of important extracts occur, we use boxes that follow a brighter version of the first box:

> **Sub-Title:** A short description.

We use the *singular they*, including the related forms *them*, *their*, and *themselves*, as the gender-neutral singular pronoun throughout this dissertation.

In case we use company names, such as Adobe, Apple, Atlassian, Google, or Microsoft, as well as product names, such as Bitbucket, Confluence, Jira, iPhone, or iPad, it is understood that these names are registered trademarks. Whenever possible, we provide links in order to find more insights using a footnote.

Part I

Prelude

As part of the PRELUDE, we introduce the research goal of this dissertation. This includes a description of our motivation and the problem context. We present our contributions in terms of design problems and knowledge questions following the design science methodology proposed by Wieringa [310]. The limitation of the research scope as well as an overview of the dissertation structure help to put the presented work into context.

This is further supported by the presentation of relevant foundations. On the one hand, we introduce important concepts on which we rely on throughout this dissertation, such as tacit knowledge, conceptual models, and user feedback, as well as various engineering and management processes, which we strive to extend. We provide general definitions and adapt them to the context of this dissertation. On the other hand, we outline important methodologies that we applied to create and validate design solutions; this comprises the design science methodology and technical action research, as well as empirical research and measurement strategies.

Chapter 1

Introduction

> *"Since the user's needs and/or the operational environment may not be fully knowable in advance, the software engineering process is at least partly a learning process. This may require that the professionals learn the users' needs, embody those needs in a requirements definition, and modify this definition as more knowledge is gained during design and implementation."*
>
> — WATTS S. HUMPHREY [146]

Software users represent the major source of knowledge for software evolution. Their demand and feedback drive the requirements engineering process of software, in particular by addressing two classes of requirements [45]:

1. **Functional requirements**, i. e., the interactions between the user and a software system to achieve an outcome on the basis of an input.
2. **Nonfunctional requirements**, i. e., the way a functionality shall be realized, such as with respect to its usability, reliability, or performance.

Both requirement classes are important for developers, as they strive for a complete understanding of the problem context [72]. However, the attempt to obtain a complete specification of user requirements for a software system often leads to a state of analysis paralysis, i. e., the actual development is postponed or never accomplished [44]. This is why the users' role in determining and defining requirements remains in the focus of ongoing research. Section 1.1 motivates the importance of users' knowledge for software engineering with two examples and points to Continuous Software Engineering (CSE) as a current approach to incorporate user feedback. Section 1.2 identifies two problems regarding the users' knowledge to describe the problem context. Section 1.3 summarizes the resulting goals this dissertation aims to resolve, while Section 1.4 lays out the research process and contributions this dissertation achieved. Section 1.5 defines the scope and Section 1.6 provides an overview of this dissertation.

1.1 Motivation

Mobile applications became an important part of our everyday life [39]. There are countless applications available and users utilize them in a multitude of ways, dependent on their context [83]. Over the last years, the progress in hardware development has enabled new software innovations and vice versa. For instance, with the pinch-to-zoom gesture that became a commodity with the introduction of the iPhone in 2007[1], the availability of technology enabled a new interaction model for mobile applications: The gesture allows to increase or decreased the zoom level of content such as images, depending on the position of the users' fingers on screen.

Users adapted and internalized this concept. It became part of their expertise in using mobile applications, as they learned to apply this interaction pattern to adjust zoom levels. However, providing an exact description of this interaction, its actual execution, the relationship between the fingers' movements and the zoom level, is hard for users to describe; an explanation of the inner workings of the gesture are, if any, best provided by a demonstration. Polanyi refers to this expertise of users as *tacit knowledge* [248]: After users have learned that one action—or a combination of such—leads to a result, they continue to apply this knowledge, without being aware *how* they are doing it, or even *that* they are doing it.

> **Example 1:** The internalization of the pinch-to-zoom gesture as tacit knowledge becomes obvious from its transfer to map-based applications: Users apply the same concept of adjusting the zoom level of images to the zoom level of maps. In combination with using the finger to move the map, this is a simple yet powerful gesture, as it allows an intuitive interaction with a map view.

The pinch-to-zoom gesture requires two hands: while users hold the device in one hand, they use the other hand to perform the gesture. This can pose a challenge, as users on the go might not have a second hand free. The need for a zoom gesture that only requires one hand evolved. Users might have verbalized this need in form of feedback to the developers, such as through a feature request.

> **Example 2:** Current map-based applications allow to adjust the zoom level of maps with only one hand: Google Maps on iOS requires users to first double tap anywhere on the screen; then, while keeping the finger on the screen after the second tap, they can increase or decrease the zoom level by sliding the finger up or down. While Google Maps zooms in when sliding up, Apple Maps on iOS does it the other way around. Also, Google Maps uses the screen center for applying the zoom, while Apple Maps relies on the position of the double tap.

[1] https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/

Example 1 highlights that users bear knowledge on how to interact with applications which they are unable to explain, however, are able to reapply and thereby transfer to other applications. Example 2 underlines that two implementations of a functionality that aim to fulfill the same requirements can have different manifestations. It is on the developers to implement a requirement in a system image that allows the users to create an applicable mental model for the functionality [226].

Current software engineering processes rely on agile methods to ensure that a feature under development follows the users' needs. A recent trend is CSE [41, 109], which is a software development process that aims to create rapid and frequent releases of a software that are delivered instantly to users. Thereby, the users always provide feedback on the latest changes. The developers retrieve user feedback that they collect from various sources, such as from software distribution systems [55, 236, 237] or social media platforms [131, 132]. User feedback such as reviews or bug reports represent *explicit* user feedback as they formalize the users' thoughts themselves [174, 223]. This allows developers to improve the software according to the users' needs. Moreover, instead of relying on a few major version, CSE promotes the development of multiple small increments that introduce minor changes. These frequent and rapid changes foster other forms of user feedback, given their nature of unobtrusiveness when collecting insights: *Implicit* user feedback, i. e., monitoring users' interactions to derive insights, promises to be a valuable source of information for user understanding. On the basis of Figure 1.1 and the following descriptions, we illustrate how user feedback in CSE affects the evolution of a user interface.

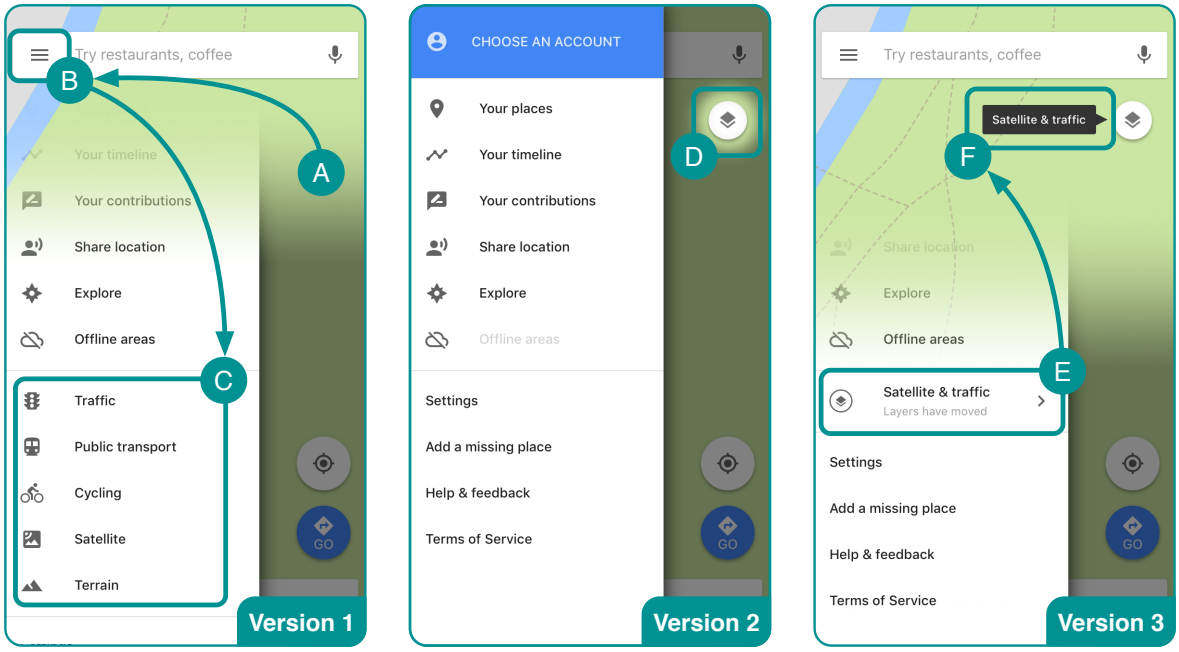


Figure 1.1: Combined screenshots depicting three versions of how to alter the map layer within the Google Maps application on iOS; screenshots from July 2017.

Example 3: Google altered the selection process of map layers Ⓐ in their iOS maps application as sketched in Figure 1.1. In *Version 1*, the selection of available map layers was accessible to users as part of the side menu Ⓒ after a tap on a so-called *hamburger* button Ⓑ. In *Version 2*, a dedicated, circle-shaped button Ⓓ was introduced which is accessible from within the map view. In *Version 3*, the developers decided to re-add a menu entry, however, this time it served as a pointer to the newly introduced button. After a tap on the menu item Ⓔ, the side menu vanished and a tool tip Ⓕ next to the new, circle-shaped button appeared.

Let us speculate about the design decisions and their rationale that preceded *Version 2* and *3* in Example 3. Most likely, the circle-shaped button introduced with *Version 2* targeted an improvement of *Version 1*. A major disadvantage could be found in the fact that the hamburger button was inaccessible during certain application usage, such as during navigation. As *Version 2* was made available to users, the developers assessed the acceptance of the newly introduced changes. We assume three key insights that were collected. First, the developers may have observed a drop in feature usage, as the users were no longer able to locate the functionality to alter the map layers. Second, by monitoring the interaction patterns of users, the developers may have detected an increased number of taps and view changes, without any resulting action—which can be interpreted that users are in the process of searching for a functionality. Third, users may have complained about the lack of the menu entry and the allegedly missing functionality. As a result, *Version 3* may have re-introduced the menu to provide a temporary solution: while retaining the improved functionality, the menu item served as a means to advert the change.

Example 3 shows how CSE allows to incrementally present minor changes to users in order to retrieve feedback that aims at understanding the users' needs better. The feedback can be of various manifestations: while written or verbal feedback depict a rich source of information, insights that are derived from user monitoring, such as observed interaction patterns or any other data from hardware and software sensors, can become relevant when trying to understand and interpret the users' needs as well as providing a context to it.

1.2 Problem Context

During the usage of software, users frequently encounter failures, i. e., a *"deviation of the observed behavior from the specified behavior"* [45]. This can be caused by a fault, i. e., a bug, which is the source for an erroneous state [45]. Likewise, users may struggle with the interfaces at hand as it does not align well with their mental model [226]. Eventually, users may develop new ideas on how the software could be improved or extended to better fit their needs, i. e., a new functional requirement.

However, research found that multiple reasons exist why users only sparely verbalize these insights. One reason may be found in their limited motivation to provide explicit feedback [8]. In particular, processes that are not fast and easy have been shown to increase the users' reluctance to provide feedback [278]. In addition, some users perceive providing feedback as a complicated process [274], especially when they are in a mobile context in which providing textual feedback requires more effort than in a desktop environment. As a result, the users do not remember problems or their initial feedback when they are being asked for it [49]. This can become particularly relevant for minor usability problems, which users may not consider important to report.

In addition, users might not be aware that they know something that is of value for the developer: When reflecting on the communication of a design, Erickson acknowledges that *"users understand, at least tacitly, their own needs and daily experience, but they understand little about the proposed design"* [94]. Sometimes, users do not know how to share, verbalize, or articulate their feedback; this can be caused by a lack of options provided to users, such as a feedback form or similar. Some users might also have no interest in sharing insights with the developers [8], while on the other hand, dissatisfied users tend to provide more feedback than satisfied users [216]. Other users might not have time to give feedback, which renders the simplicity and unobtrusiveness of the feedback process as a critical factor whether they share insights with the developers or not [8]. While users' insights are not limited to the detection of bugs or misuses, there is also the case that users can act as providers of new features. This is reported as a feature request, while similar problems as the ones mentioned above apply. We summarize this as follows:

> **Knowledge Extraction Problem:** Software users carry tacit knowledge which is not shared with software developers.

CSE acknowledges this problem by applying an agile and iterative approach [41, 109]; it aims to closely involve the user in the development process to retrieve feedback early. However, most of the CSE processes rely on traditional feedback capture, which leads to inefficiency, as retrospective approaches tend to be time consuming and dependent on individuals [196]. As a result, feedback is treated in a discrete approach, rather than a continuous source of feedback.

Generally, it appears that the interaction with user feedback is less mature and less formalized compared to activities such as continuous delivery or deployment: In a systematic mapping study with 50 primary studies, Rodríguez *et al.* found that, with one exception, *"mechanisms for systematically processing feedback were not elaborated in the primary studies"* [258]. Likewise, Bano and Zowghi found that only a few studies report on results and even less on successful integration of user involvement into requirements engineering activities [23].

With respect to agile methods such as CSE, Seffah *et al.* emphasize a lack of guidance when it comes to the validation of user requirements [283]. Chilana *et al.* note that, while quantitive data from usage monitoring help to understand user behavior in large numbers, it does not provide a reason for that particular behavior [61]. They argue that it is the need to adapt existing user testing methods to CSE and agile practices and emphasize efforts for time and cost [61]. Pagano describes a *social beta* approach, in which user feedback clusters are used to determine the quality of small and frequent feature increments [235]. We remain with the observation that user understanding has not yet reached a maturity level within CSE required to extract users' tacit knowledge. We summarize this as follows:

> **Knowledge Integration Problem:** Tacit knowledge that resides within users is not systematically integrated into development processes such as continuous software engineering.

Our goal is to change that. This requires answers to questions such as how to deal with a small number of users during an early software stage, how to utilize implicit usage data to overcome the limited availability of user feedback, or how feedback can be related to features under development.

1.3 Goals

As it becomes obvious from the problem context, this dissertation starts with a focus on two main stakeholders: user and developer of mobile software applications. The user has an interest in the project given that they desire software that addresses their needs. As we research tacit knowledge, not only are users not aware that they are a stakeholder, they also do not know that they have an interest, i. e., a goal, which is to be provided with software that better addresses their needs. The developer's goal is driven by these goals of the users; as the creator of an application, they want to satisfy the needs of the users in form of features that fulfill both functional and non-functional requirements. In the long run, this ensures that users continue utilizing their applications, which, for example, results in a positive effect for an enterprise or company. The long-lasting motivation of successfully, effectively, and sustainably running a business, however, is out of scope for this dissertation.

We focus on the relationship between developers and users. We strive for a *continuous user understanding activity* to enable the comprehension of users with the goal to refine existing and specify new functional and nonfunctional requirements.

> **Research Goal:** Establish continuous user understanding so that developers can extract tacit knowledge for utilization in software evolution.

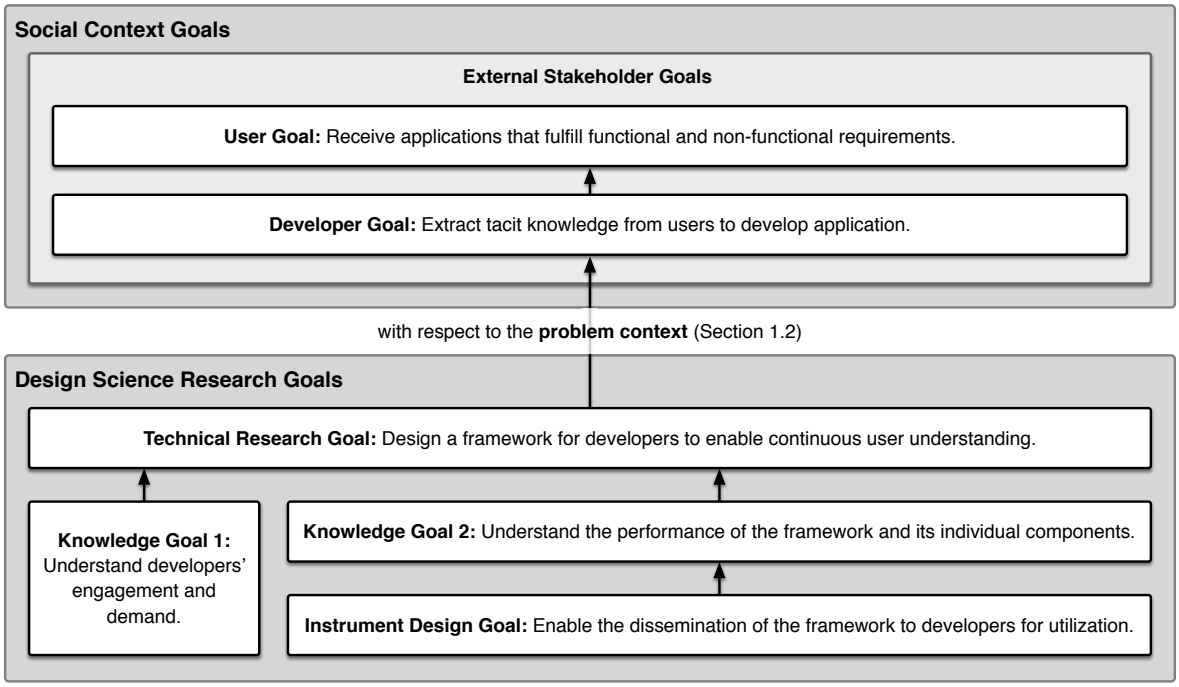


Figure 1.2: Goal hierarchy adapted from Wieringa [310], with instrumented design goals on the lowest level. An arrow indicates that a goal supports the other.

We base our research goal on established requirements engineering processes. Our overall vision is to implicitly derive requirements from users in the first place: We envision an approach in which the developers start with an initial understanding of a potential future user of a system, a synthetic user or persona of a future user, and—by incrementally releasing software increments—asymptotically approach the users' needs based on a mixture of knowledge sources.

According to Wieringa, a design science research project typically consists of a goal hierarchy [310], which we summarize in Figure 1.2 and explain in the following. Our research goal for developers is to increase software quality and to be able to create a more accurate fit of functionality for users' needs. Continuous user understanding aims to satisfy the *external stakeholders' goals*, which are part of the *social context goals* and defined by the *problem context*. Thereby, the research project goals are ultimately driven by the stakeholders' goals. We explore the stakeholders' goals in detail with an interview study described in Part II.

To achieve our research goal, the design science research goals that are bundled by this dissertation require to investigate on artifacts, i. e., software systems and concepts, that drive and support the achievement of the continuous user understanding activity. We summarize this with the following Technical Research Goal (TRG):

> **Technical Research Goal:** Design a framework for developers to enable continuous user understanding during continuous software engineering.

We achieve this goal with **CuuSE**, a framework[2] for continuous user understanding in software evolution. The framework bundles a set of artifacts that can interact with and rely on each other. To be able to understand its requirements and implement the artifacts, we need to further clarify the needs of developers and users. We summarize this effort with the following Knowledge Goal (KG):

> **Knowledge Goal 1 ("Investigation"):** Understand developers' engagement and demand during continuous software engineering.

At the same time, we want to understand how the **CuuSE** framework and its artifacts, which result from the technical research goal, solve the knowledge extraction and integration problems. Therefore, we address another knowledge goal:

> **Knowledge Goal 2 ("Validation"):** Understand the performance of **CuuSE** and its individual components during continuous software engineering.

To enable the validation, we investigate the **CuuSE** framework in an environment that is as realistic as possible. This means an application of **CuuSE** within a multi-project setting that is almost similar to industrial environment. The framework needs to be made understandable for developers to be useful. This is reflected in what Wieringa calls the Instrument Design Goal (IDG): *"An instrument design goal is the problem to design an instrument that will help us answer a knowledge question"* [310], which we instantiate as follows:

> **Instrument Design Goal:** Enable the dissemination of **CuuSE** to developers for its utilization during continuous software engineering.

1.4 Contributions

"Goals define problems" [310]—we interpret Wieringa's meaning of *defining* in a way that we can have an initial understanding of a set of problems to describe their bounding boxes. We identify four technical research problems for which we present treatments in this dissertation; we describe them in Section 1.4.1.

While we can describe a first version of the problems beforehand, we need to gain a better understanding of the social and knowledge context as part of the problem-solving process [310]. The investigation on knowledge goals, which *"should be refined into knowledge questions"* [310], drives the design of the artifacts, which are the manifestations of the treatment proposed for the problems. We identify five knowledge questions and describe them in Section 1.4.2.

[2]Following the definition of Bruegge and Dutoit [45], we consider **CuuSE** a framework, as it can be used and extended for other purposes. The adaption to other domains is outlined in Section 13.4.

1.4.1 Design Problems

From the technical research goal, we derive Technical Research Problems (TRPs). For their definition, we follow the template proposed by Wieringa [310]: we highlight artifacts, requirements, and stakeholder goals. We omit the problem context, which remains the same for all problems and is described in-depth in Section 1.2.

First, we investigate on how to provide developers with a means to explore usage knowledge. We present the **Cuu** workbench, i. e., the continuous user understanding workbench, which comprises a dashboard for knowledge visualization and a platform that integrates with a CSE process. The **Cuu** workbench allows developers to reflect and interact with usage knowledge. We summarize this as follows:

> **Technical Research Problem 1:** How to implement a workbench that integrates with continuous software engineering and visualizes knowledge so that developers can access usage knowledge?

Second, we address traceability of usage knowledge to a feature. Typically, usage knowledge is collected only on the basis of a release. However, in particular for CSE processes which frequently release increments containing only minor changes, a more fine-grained concept is required. We introduce feature crumbs, a lightweight, code-based concept to specify a feature's run-time characteristics. Feature crumbs allow to precisely map usage information to observations and create a context which allows the comparison of usage knowledge. We summarize this as follows:

> **Technical Research Problem 2:** How to design a monitoring concept that enables a mapping between usage information and an increment's particulars so that developers can investigate on their relationship?

Third, in order to enrich the **Cuu** workbench with knowledge, we investigated the gathering of usage knowledge through knowledge sources. The knowledge sources contribute to derive knowledge that is deeply ingrained in the users' minds and which is difficult for them to verbalize. We refer to these knowledge sources as **Cuu** kits. They must allow unobtrusive knowledge monitoring, which we define by being able to either collect data implicitly, or by allowing for a highly scalable and automated collection and analysis process. We present four kits that fulfill this requirement, namely *EmotionKit*, *BehaviorKit*, *ThinkingAloudKit*, and *PersonaKit*. We summarize this as follows:

> **Technical Research Problem 3:** How to implement knowledge sources that allow the unobtrusive collection of usage knowledge so that developers can retrieve a variety of insights for user understanding?

Fourth, developers need to be advised with a good practice on how to apply the **Cuu** workbench in combination with the utilization of the results from different **Cuu** kits. This includes the integration of feature crumbs, which directly effects the quality of the results being presented to the developers. We designed the **Cuu** workflow, which represents a critical aspect of continuous user understanding, as the dynamic aspect eventually enables the extraction of tacit knowledge. We summarize this as follows.

> **Technical Research Problem 4:** How to design a workflow that combines a workbench, a monitoring concept, and different knowledge sources for continuous user understanding so that developers can extract tacit knowledge?

The result of these technical research problems are multiple artifacts, i. e., the **Cuu** workbench, the feature crumbs concept, a set of **Cuu** kits, and the **Cuu** workflow, which represent building blocks of the **Cuu**$^{\textbf{SE}}$ framework.

1.4.2 Knowledge Questions

To achieve Knowledge Goal 1, we performed a semi-structured interview study with practitioners, using which we address three Knowledge Questions (KQs).

First, we investigated on how CSE is currently being applied by practitioners. Understanding their definition of CSE, most relevant elements for CSE, their experiences, and plans for further additions to their CSE process forms the basis for finding a problem solution and seamlessly integrate a continuous user understanding activity. We summarize the related question as follows:

> **Knowledge Question 1:** How do practitioners apply continuous software engineering during software evolution?

Second, we were interested in understanding current practices of how practitioners involve and rely on users to understand their needs. Therefore, we asked the practitioners on how they consider, capture, and utilize user feedback. The results provide insights on how to organize and design features of the **Cuu**$^{\textbf{SE}}$ framework. We summarize the related question as follows:

> **Knowledge Question 2:** How do practitioners involve users during continuous software engineering?

Third, we continued to explore practitioners' understanding of CSE by asking them about the integration of different knowledge types into the CSE processes. We proposed a vision of knowledge integration into CSE to practitioners, which we utilized as the foundation for addressing the technical research problems. We collected

detailed insights on practitioners' general attitude toward knowledge integration, as well as their opinion of perceived benefits, obstacles, important extensions, and potential additions. We summarize the related question as follows:

> **Knowledge Question 3:** How does usage knowledge support practitioners during continuous software engineering?

Following the design cycle described by Wieringa [310], the design of new solutions for existing problems as part of the treatment design usually results in new knowledge questions. Therefore, as part of the treatment validation, we defined two additional knowledge questions that address the effects of the treatment artifacts. Using the following knowledge questions, we address Knowledge Goal 2.

First, we validated the treatment for the Technical Research Problem 3, i. e., the implementation of **Cuu** kits. We put an emphasis on the validation process of one kit, i. e., EmotionKit, and outline initial validation results for BehaviorKit, ThinkingAloudKit, and PersonaKit. We investigated on their trade-offs and sensitivity effects. We devote the following knowledge question to the validation of the **Cuu** kits:

> **Knowledge Question 4:** How do knowledge sources perform as a means to collect usage knowledge?

Second, we validated the simultaneous application of all treatments for the Technical Research Problems 1, 2, 3, and 4, since by combining the individual artifacts, the dissertation's research goal of continuous user understanding should be established. We performed two technical action research cycles. During one helper cycle, we collected developers perceived usefulness, perceived ease-of-use, and the intention when interaction with **Cuu**$^{\text{SE}}$ in the problem context. We summarize the related question as follows, while its results induce the need for future development of **Cuu**$^{\text{SE}}$ and promote future work:

> **Knowledge Question 5:** How does the **Cuu**$^{\text{SE}}$ framework support developers during continuous user understanding?

To answer Knowledge Question 5, we were first required to address the Instrument Design Goal. Therefore, as part of another helper cycle, we designed and validated the **Cuu** syllabus as a treatment to enable the validation of the **Cuu**$^{\text{SE}}$ artifacts. Following the terminology of technical research problems and their respective goal, we summarize the related Instrumental Design Problem (IDP) as follows:

> **Instrumental Design Problem:** How to design a syllabus that enables the dissemination of **Cuu**$^{\text{SE}}$ so that researchers can validate the framework?

1.4.3 Research Cycles

Together with the researcher's interest, the problem context triggers the engineering cycle [310]. We illustrate the methodological approach used in this dissertation in Figure 1.3. The problem context also plays an important part in the problem investigation; during this part of the engineering cycle, the problem context should be investigated and explored in detail.

During problem investigation, we resolved Knowledge Goal 1 using the *Empirical Cycle 1*. We conducted a semi-structured interview study with 24 practitioners. We decided to focus on the analysis of state-of-the-practice empirical cycles because the problem context of CSE is mainly driven by industry. We resolve knowledge questions regarding the phenomena of tacit knowledge and related state-of-the-art in the respective sections of the technical research problem.

During the treatment design phase, we worked on the Technical Research Goal and developed four artifacts, i. e., a workbench, a monitoring concept, a set of knowledge sources, and a workflow. The work on artifacts led to new knowledge questions regarding requirements and existing treatments. While the related empirical cycles are not visualized in Figure 1.3, we answer them through literature reviews.

The treatment validation was concerned with Knowledge Goal 2, as we try to understand how the artifacts perform in the problem context. We dedicate one cycle, *Empirical Cycle 2*, to validate the *EmotionKit* knowledge source in-depth, using a laboratory experiment with 12 participants. Nevertheless, to answer the knowledge questions for all knowledge sources, we applied an analytical approach which is not depicted in Figure 1.3. Therefore, we compared multiple knowledge sources from multiple perspectives, i. e., in terms of trade-off and sensitivity questions.

To complete the validation of the **Cuu^{SE}** framework, we initiated the *Empirical Cycle 3*, which comprises two client cycles: using a technical action research approach, we focus on how **Cuu^{SE}** supports developers during software evolution.

Client Cycle 1 investigated the problem of how to enable the dissemination of **Cuu^{SE}** to developers. We addressed the Instrument Design Goal to answer the second knowledge goal. We designed the **Cuu** syllabus and validated it using a quasi-experiment with 9 students during the iPraktikum, a multi-project university course as described in Section 11.2, of the summer term 2018. *Client Cycle 2* represents another instance of the iPraktikum, i. e., of the winter term 2018/19. We used a survey in order to understand the developers' attitude toward the **Cuu^{SE}** framework. Therefore, we designed a combination of the **Cuu^{SE}** framework and the **Cuu** syllabus.

For both client cycles, as well as for the overall engineering cycle, the evaluation and implementation of the treatment is out of scope for this dissertation, which is indicated through the transparent elements in Figure 1.3. However, the client cycles provide many insights which could be used to start further cycles for work on the treatment design and initiate future work.

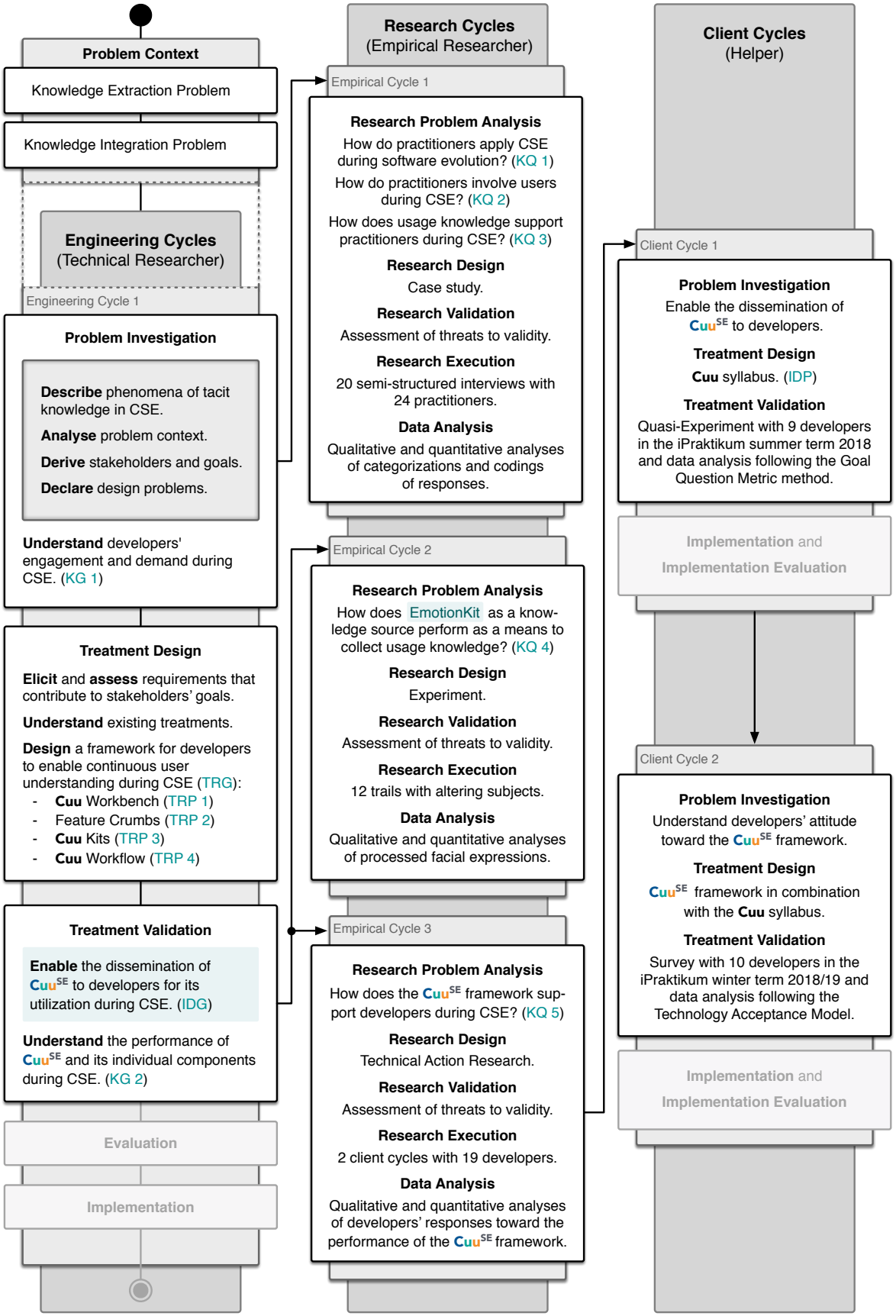


Figure 1.3: The engineering, research, and client cycles based on Wieringa [310, 311].

1.5 Research Scope

The analysis of user feedback is an established area of interest for research in software engineering; progress in CSE and new techniques from the domain of machine learning are adding new stimuli. We limit this dissertation as follows.

User Samples

This dissertation puts an emphasis on developing a continuous user understanding activity during the early development of an application, which typically involves major and drastic refinements in terms of what an application offers or how it can be operated. This foundational principle is highlighted by Nielsen, who proposes to test with only five users [219]. We further stress this aspect by referring to the main **CuuSE** artifact as a workbench and intend to draw an analogy to a craftsmanship. Therefore, we focus on user feedback that can be drawn from a finite population size. In other words, we do not consider techniques that employ statistical tests, which in turn would require uniformly distributed population samples which require a complete population. It should be noted that these might be integrated as a knowledge source, and that the **CuuSE** framework generally supports a combination.

We do not track an individual to understand *this* particular person. This draws a clear line to analytics that target marketing purposes. We are interested in the character traits, thoughts, and impressions, rather than in the individuals themselves. There is a dilemma in being able to track an individual between software releases, as it removes the ability to compare user feedback between two releases. While still making it possible to compare results between releases, we approach this by acknowledging that we might compare results from to different persons.

Finally, with respect to both of these aspects, we are limiting our analyses to users that are aware of being monitored. They have been informed before testing the feature under development. This might have the effect on the validity of the knowledge sources' results. This reflects an important aspect for privacy, which we address throughout our work and briefly summarize in Section 13.2.

Consumer Data Sensors

The knowledge sources are built from collector components that monitor different user characteristics to process observed usage data to usage information and knowledge. We focus on the use of data sensors that are available to a broad audience of users, i. e., sensors that became a *commodity*. The most basic sensors are software sensors; from a hardware perspective, we consider sensors such as microphones or various types of cameras as a consumer data sensor. We exclude medical devices or sensors that are not usually worn or present in a day-to-day application scenario.

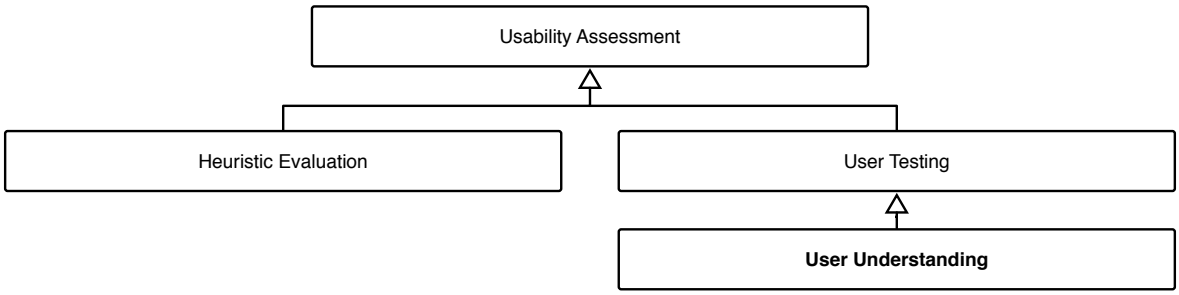


Figure 1.4: User understanding beyond heuristic evaluation and user testing.

Evidence-Based Knowledge Sources

In contrast to opportunistic approaches, we are not randomly releasing a version to understand and investigate the fine nuances in the change of users' behavior. We aim to develop an evidence-based model that follows the asymptotic convergence of the feature based on qualitative, well-defined, considered, and justified reasons. This further contrasts the approaches used in **Cuu**$^{\text{SE}}$ to methods such as A/B tests, or generally concepts from continuous experimentation. This way of approaching user understanding is acknowledged by companies, such as Facebook, that do not solely rely on A/B tests, but also perform in-house usability tests in addition to their tests on a larger scale after an increment has been deployed [102].

In particular, web applications are increasingly approached with automatic assessment methods [202, 256]. As we intend to derive our insides from actual user interaction, we build on the foundations of user testing for the approaches presented in this dissertation; as shown in Figure 1.4.

Software Types

First, we limit the applicability for **Cuu**$^{\text{SE}}$ to applications that rely on a graphical user interface; which implies that they require physical input by the users rather than other input sources, such as voice control. Second, we focus on mobile applications as the target environment in which **Cuu** kits are deployed.

Design- and Runtime Usage of Feedback

We focus on the utilization of user feedback during the design time of an application; this means that, while the information collected by knowledge source may be used to alter the user interface during the runtime of an application, we focus on the use of information for the developer to make design time modifications. In addition, we strive to improve the application from a usability perspective, including the need to improve or add new features; we do not focus on the detection of issues in code that are generally considered a bug, or any other performance analysis.

1.6 Dissertation Structure

This dissertation is structured into five parts. In line with the design science methodology by Wieringa [310], the dissertation's main contribution are reflected in Parts II, III, and IV, which adhere to the naming structure of *Problem Investigation*, *Treatment Design*, and *Treatment Validation*. While the two parts problem investigation and treatment validation address the knowledge questions that we introduced in Section 1.4.2, the treatment design focuses on the technical research problems that we introduced in Section 1.4.1. In this dissertation, we enclose the engineering cycle, that consists of the three phases, by a *Prelude* as Part I (current part) and a *Postface* as Part V. We present an overview of the dissertations' structure including respective results, addressed knowledge questions and technical research problems, in Figure 1.5. In the following, we summarize the content of all remaining chapters.

Chapter 2 presents foundations that are relevant for understanding the context of this dissertation. This includes an introduction to tacit knowledge, which represents knowledge deeply ingrained in the individuals' mind; we transfer the concept to software engineering with the notation of conceptual models and introduce a taxonomy of user feedback. We clarify important processes such as requirements and usability engineering. In addition, this chapter establishes a common understanding of research methodologies, i. e., design science and technical action research, strategies for empirical research, i. e., case study, survey, and experiments, as well as strategies for measurement, i. e., the goal question metric approach and the technology acceptance model.

Chapter 3 lays the foundations for a comprehensive case study in which we relied on semi-structured interviews to address multiple knowledge questions. We start the chapter by introducing our understanding of CSE and the introduction to the CURES project, which served as the higher-level research scope for this dissertation. With respect to the study design, we elaborate on its structure, such as the case and subject selection, the data collection, and the analysis procedure, as well as descriptive data and the validity of the case study.

Chapter 4 reports on the excerpt of the case study that addresses practitioners' perception of CSE. Therefore, we asked them about their personal definition of CSE, elements of CSE which they perceive as the most relevant, their experiences with the application of CSE elements, as well as their future plans for CSE. We provide both a quantitative and qualitative presentation and a discussion of the results.

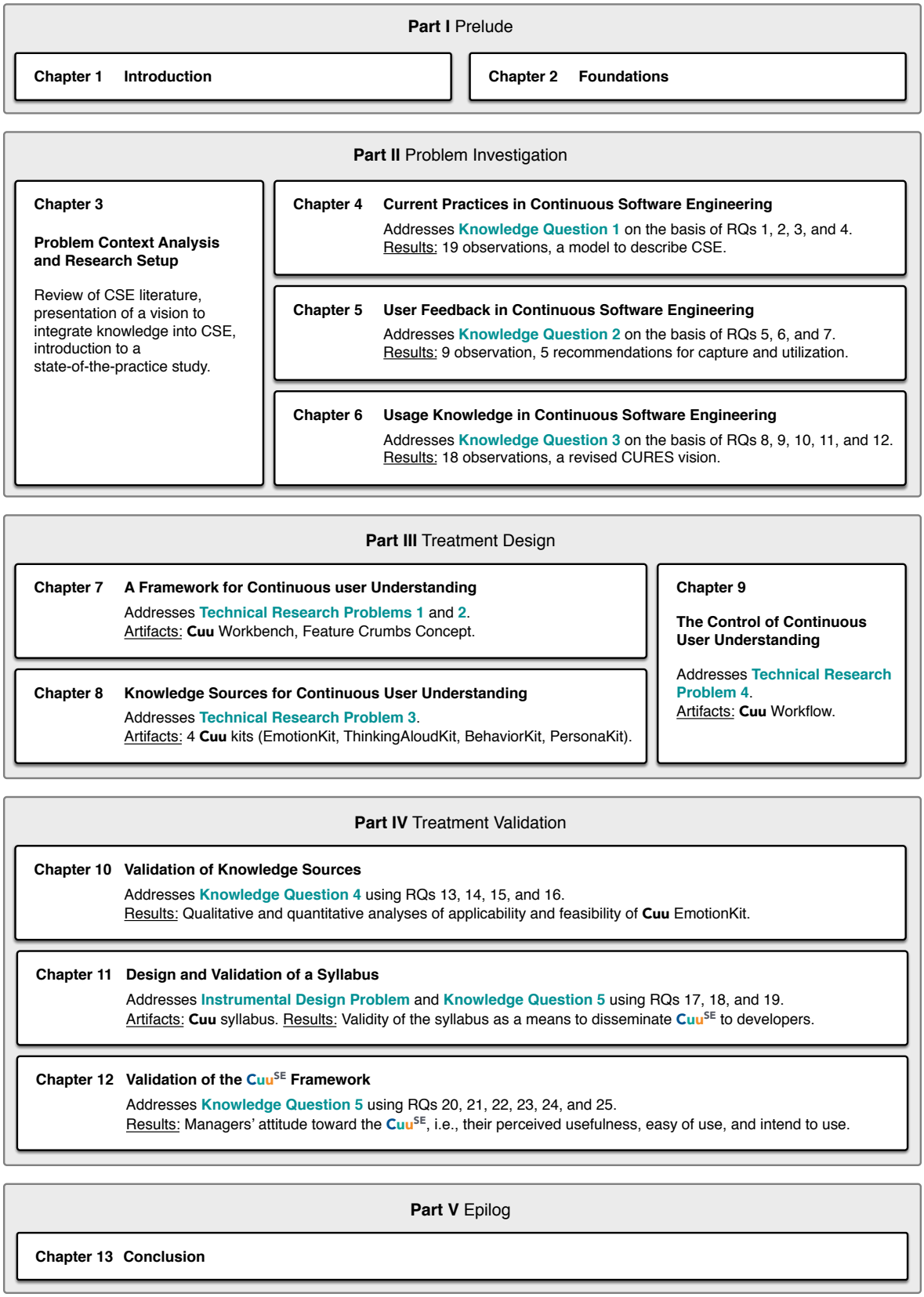


Figure 1.5: Overview of the dissertation structure based on Wieringa's Design Science approach [310].

Chapter 5 reports on the excerpt of the case study in which we investigated how practitioners involve users during CSE. Therefore, we asked the practitioners which user feedback they consider as well as how they capture and utilize user feedback in CSE. We provide both a quantitative and qualitative presentation and a discussion of the results.

Chapter 6 reports on the excerpt of the case study that investigates how usage knowledge supports practitioners during CSE. Therefore, we introduced the practitioners to a vision originated from the CURES project that proposes the integration of knowledge into CSE. We asked the practitioners for their attitude toward the vision, as well as perceived benefits, obstacles, extensions, and additions. We provide both a quantitative and qualitative presentation and a discussion of the results.

Chapter 7 depicts an architecture design, that follows a blackboard pattern, to enable continuous user understanding. We introduce two main artifacts, namely a usage monitoring concept, that enables the mapping of usage knowledge the feature particulars, and a workbench, that enables the visualization of collected usage knowledge. We elaborate on both artifacts by presenting related treatments, eliciting requirements and presenting implementation details.

Chapter 8 outlines four knowledge sources that contribute to the knowledge that can be accessed through the workbench, namely to extract emotions from facial expressions, to automate the collection of verbalized thoughts, as well as to classify interaction data for individual users and groups of users. We provide implementation details on all knowledge sources and detail a software development kit for developers.

Chapter 9 completes the presentation of artifacts with the introduction of a workflow that enables developers to make use of the workbench, the usage monitoring concept, and the knowledge sources. We take a closer look on how usage knowledge is analyzed by developers as well as provide a discussion and analysis of related workflows.

Chapter 10 begins with the validation of the presented artifacts. In this chapter, we focus on a detailed validation of EmotionKit as a representative for all knowledge sources. We outline the experiment design, report on results of a qualitative and quantitative analysis, and discuss our findings. We close the chapter with a brief overview of how the remaining knowledge sources can be validated individually, as well as with suggestions for an interrelated validation that considers all knowledge sources.

Chapter 11 addresses the design and validation of a syllabus. We start by discussing its design and detail its structure; as the dissemination of the developed artifacts require further effort, the syllabus serves as a means to introduce student developers to the **Cuu^{SE}** framework. We outline a university capstone course, which formed the environment for the validation of the syllabus. Eventually, we provide the results and interpretations of the validation of the syllabus during one semester.

Chapter 12 deals with the combined validation of all artifacts that were introduced within the scope of this dissertation. We report on a survey with student developers in the context of the university capstone course introduced in the previous chapter. Their answers regarding their perceived usefulness, their perceived ease of use, and their intention to continue using **Cuu^{SE}** reflect important insights on the acceptance of the framework and provide starting points for future work.

Chapter 13 concludes this dissertation. First, we provide a short summary of the main contributions. Second, we outline the relevance of privacy aspects in the context of the dissertation's topic. Third, we suggest the improvement of the presented framework toward automatic tacit knowledge extraction. Fourth, we reflect on future extensions of **Cuu^{SE}** to other domains, such as mobile and smart environments.

This dissertation is based on multiple publications, which are listed in Table 1.1. The first column lists the **Publication**'s meta information, such as authors' names. The second column **Ch.** indicates the respective chapter in which their content was primarily integrated; minor contributions of a publication might be scattered throughout the dissertation.[3] In the third column **Details**, additional publication meta data is presented, i. e., the publications' Digital Object Identifier (DOI), the publisher, and copyright details. In case a DOI is not available, a link to an online repository is provided. Only the publisher name is stated, more information and detailed copyright guidelines are provided in the respective section of Appendix A. In Table 1.1, *editor* denotes exceptions for work that was published via the CEUR Workshop Proceedings.[4] With respect to the publications listed in Table 1.1, it should be noted that in particular publications [157, 158] are the result of a joint work with Heidelberg University as part of the CURES project (Section 3.2).

In addition to these publications, several theses and research projects further contributed to this dissertation [33, 34, 108, 114, 129, 254, 319].

[3] This reflects the fact that the individual publications are placed and refined in a broader context.

[4] *"The publishers of the proceedings volumes (CEUR-WS.org/Vol-1, CEUR-WS.org/Vol-2, etc.) are the respective editors of the volumes."*—CEUR Workshop Proceedings information text, available online at http://ceur-ws.org.

Table 1.1: List of publications on which this dissertation is based on.

Publication	Ch.	Details
Johanssen, J. O., Kleebaum, A., Bruegge, B., & Paech, B. (2017). Towards a Systematic Approach to Integrate Usage and Decision Knowledge in Continuous Software Engineering. In: *Proc. of the 2nd Workshop on Continuous Soft. Engineering* [153]	3	Available online, editors, Appendix A.5
Johanssen, J. O., Kleebaum, A., Bruegge B., & Paech B. (2017). Towards the Visualization of Usage and Decision Knowledge in Continuous Software Engineering. In: *Proc. of the 5th IEEE Working Conference on Software Visualization* [154]	7	10.1109/VISSOFT.2017.18, IEEE, Appendix A.1
Johanssen, J. O., Kleebaum, A., Paech B., & Bruegge, B. (2018). Practitioners' Eye on Continuous Software Engineering: An Interview Study. In: *Proc. of the International Conference on Software and System Processes* [157]	3, 4	10.1145/3202710.3203150, ACM, Appendix A.3
Johanssen, J. O. (2018). Continuous User Understanding for the Evolution of Interactive Systems. In: *Proc. of the 10th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* [150]	1	10.1145/3220134.3220149, ACM, self-owned
Johanssen, J. O., Kleebaum, A., Bruegge, B., & Paech, B. (2018). Feature Crumbs: Adapting Usage Monitoring to Continuous Software Engineering. In: *Proc. of the 19th International Conference on Product-Focused Software Process Improvement* [155]	7	10.1007/978-3-030-03673-7_19, Springer, Appendix A.4
Johanssen, J. O., Henze, D., & Bruegge, B. (2019). A Syllabus for Usability Engineering in Multi-Project Courses. In: *Proc. of the 16th Workshop of Software Engineering im Unterricht der Hochschulen* [152]	11	Available online, editors, Appendix A.5
Johanssen, J. O., Kleebaum, A., Paech, B., & Bruegge, B. (2019). Continuous Software Engineering and its Support by Usage and Decision Knowledge: An Interview Study with Practitioners. In: *Journal of Software: Evolution and Process* [158]	3, 4, 6	10.1002/smr.2169, Wiley, Appendix A.2
Johanssen, J. O., Reimer, L. M., & Bruegge, B. (2019). Continuous Thinking Aloud. In: *Proc. of the Joint 5th Int. Workshop on Rapid Continuous Soft. Eng. & 1st Int. Workshop on Data-Driven Decisions, Exp. and Evolu.* [159]	8, 10	10.1109/RCoSE/-DDrEE.2019.00010, IEEE, Appendix A.1
Johanssen, J. O., Bernius, J. P., & Bruegge, B. (2019). Toward Usability Problem Identification Based on User Emotions Derived from Facial Expression. In *Proc. of the 4th International Workshop on Emotion Awareness in Software Engineering* [151]	8, 10	10.1109/SEmotion.2019.-00008, IEEE, Appendix A.1
Johanssen, J. O., Viertel, F. P., Bruegge, B., & Schneider, K. (2019). Tacit Knowledge in Software Evolution. In: *Design for Future — Managed Software Evolution* [160]	2, 13	10.1007/978-3-030-13499-0_5, Springer, Appendix A.5
Johanssen, J. O., Kleebaum, A., Bruegge, B., & Paech, B. (2019). How do Practitioners Capture and Utilize User Feedback during Continuous Software Engineering? In: *Proc. of the 26th International Requirements Engineering Conference* [156]	3, 5	10.1109/RE.2019.00026, IEEE, Appendix A.1

Chapter 2

Foundations

> *"When we go about the spontaneous, intuitive performance of the actions of everyday life, we show ourselves to be knowledgeable in a special way. Often we cannot say what it is that we know. When we try to describe it we find ourselves at a loss, or we produce descriptions that are obviously inappropriate. Our knowing is ordinarily tacit, implicit in our patterns of action and in our feel for the stuff with which we are dealing. It seems right to say that our knowing is **in** our action."*
>
> — DONALD A. SCHÖN [280]

In this chapter, we resolve potential ambiguities that may arise in the context of this dissertation; this common understanding prepares the following chapters.

In Section 2.1, we start with the clarification of the term tacit knowledge, which has been in the focus of reachers for decades and can be found in various domains. We continue how tacit knowledge can be reflected in so-called conceptual models in the context of software engineering, which leads us to a taxonomy of user feedback, on which we rely throughout the dissertation. We summarize important engineering and management processes, such as requirements engineering, usability engineering, and rationale management, and point out how they relate to the dissertation's topic of continuous user understanding in software evolution.

In Section 2.2, we introduce multiple methodologies that we applied throughout the dissertation. First, we present design science, which represents the main methodology that we used to investigate on the problem context, as well as create and validate artifacts; this includes a description of technical action research. Second, we introduce the definitions of three empirical research strategies, namely case studies, surveys, and experiments, all of which we relied on during both problem investigation and treatment validation. Third, we discuss strategies for data measurement and assessment, i. e., the Goal Question Metric (GQM) approach as well as the Technology Acceptance Model (TAM).

2.1 Concepts

In this section, we introduce and describe notations, definitions, and terms that we used throughout the dissertation.

2.1.1 Tacit Knowledge

Tacit knowledge reflects a kind of expertise that is deeply ingrained in an individual's mind [248]; the individual is able to apply such knowledge repeatedly, however, it may not be able to verbalize its particulars, i. e., what needs to be done to which extend in order to achieve a certain outcome. For example, with respect to software engineering, developers tend to design a user interface following rules of simplicity and ease-of-use when working on a development task—without an explicit requirement that asks them to do so. Most of the time, when an individual's activity of expertise is interrupted while they apply their tacit knowledge, they may be capable of verbalizing some kind of the rationale for it. However, in many cases, domain experts are not aware that their expertise depends on tacit knowledge and that it might be of use for others.

Polanyi build his definition of tacit knowledge on the fact that *"we can know more than we can tell"* [248]: In his book *The Tacit Dimension*, he coined the term tacit by describing it as a skill, locating the sense of the term closely to a physical interaction with an object, such as riding a bicycle or playing an instrument; all of these actions share the characteristic that they need to be learned over a long period of time and are typically difficult to transfer using words only. Polanyi systematically described the inner workings of a human are about to experience, i. e., when they externalize, tacit knowledge. He described this as a *functional* relationship and structure of tacit knowledge: Both allow to disassemble individual parts of tacit knowledge. In addition, he identified semantic and ontological aspects which lead to the *phenomenal* structure of tacit knowing, i. e., an externalized manifestation.

A similar description is provided by Gigerenzer, who used the analogy of a native speaker: while they are able to construct a sentence which is grammatically correct, they may not be able to verbalize the respective grammar [120]. He referred to this as a *"gut feeling"* and used the term interchangeably with intuition and hunch. Gigerenzer exemplified that humans tend to choose logically unlikely alternatives when asked for predicting the likelihood of two alternatives, the *"conjunction fallacy"*, which is based on impressions rather than mathematical rationale.

Schön recognized similar patterns in working environment settings when he describes that *"the workaday life of the professional depends on tacit knowing-in-action"* [280]. He observed that in particular professionals are knowledgable about their action and continuously make judgement on various aspects for which they are unable to state measurable and definable quality criteria; the results, however, are at

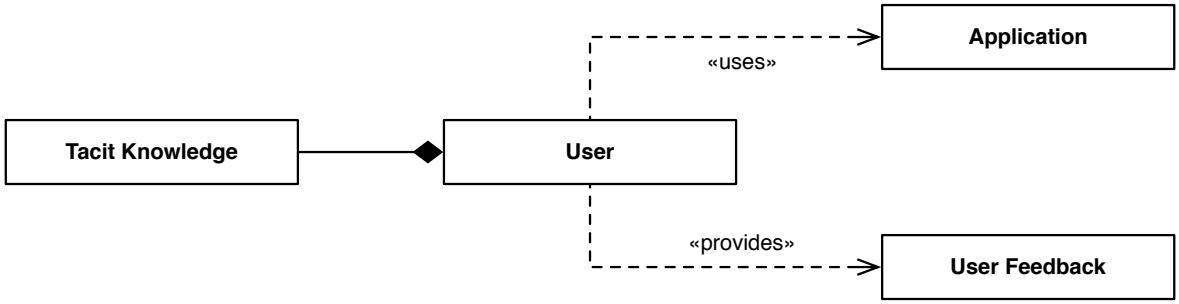


Figure 2.1: A model of tacit knowledge during software usage (UML class diagram).

expected quality. Schön argued that the extraction of *"know[ledge]-in-action"* is similar to an art, while one way to approach it is the *"reflection-in-action"*, i. e., stop doing what one was doing with the goal to reflect for a better understanding of such.

Nonaka and Takeuchi provided an extensive examination of the differences between *explicit* knowledge, i. e., anything that is written down in rules, definitions, or handbooks, and *implicit* knowledge, i. e., the experiences of an individual that are based on personal values and motivated by cultural aspects [224]. They described the dynamic interplay between these two knowledge types as the key for knowledge creation in companies. They established a spiral model that contributes to the social process of knowledge sharing that heavily depends on a collaborative interaction and leads to the externalization of knowledge which makes it useful for companies.

Tacit knowledge is studied in different contexts: For example, as previously introduced, tacit knowledge that is carried by professionals [280] or resides with companies [224]. But also other domains study the existence of tacit knowledge, for example in medicine [240]. Understanding and externalizing tacit knowledge can be valuable for other disciplines as well, in particular for software engineering. For instance, Schneider acknowledged that specific techniques are needed to capture requirements and additional information when and where they surface: in natural-language requirements specifications or by observing activities by experts [277]. Another example is provided by LaToza *et al.*, who highlighted knowledge that resides in developers' minds regarding the application of tools and activities to perform code tasks during software development [185].

In the context of this dissertation, we focus on tacit knowledge that is carried by user of an application. As outlined in Figure 2.1, we distinguish between four main entities: The `User`, who `uses` an `Application` which is represented by a software system and helps them to achieve a goal. The way they use the application is driven by their `Tacit Knowledge`. It represents the user's idea of how they expect the application to accomplish a given task and how they should behave given a certain state of the application. After and during the use of the application, they provide `User Feedback`, which we further analyze in Figure 2.3.

2.1.2 Conceptual Models

To capture the concepts of a user's tacit knowledge, Norman coined the term *Conceptual Model*, which aims to formalize the description of different perspectives on a similar issue, artifact, or problem at hand [226]. In Figure 2.2, we sketch involved models in the context of software engineering.

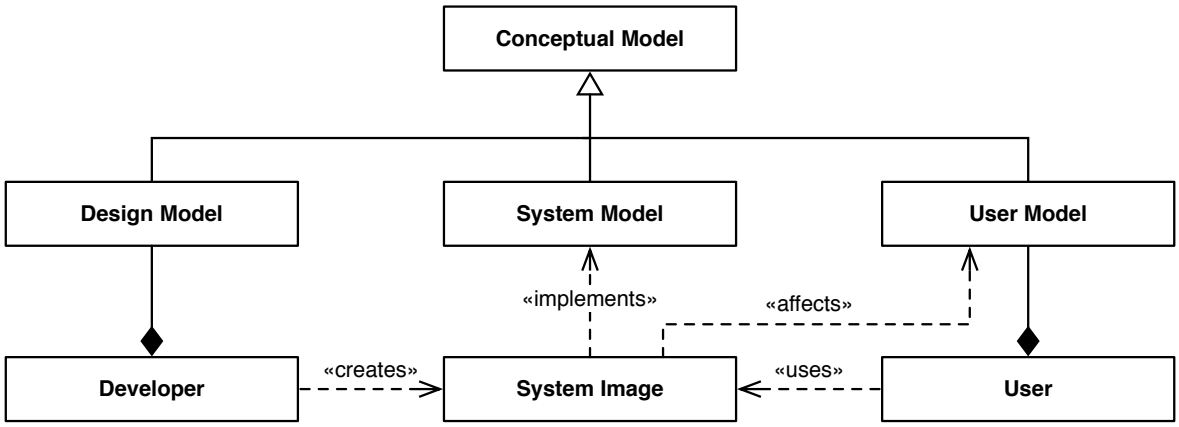


Figure 2.2: Conceptual models as described by Norman [226] adapted to the context of software engineering (UML class diagram).

The `Conceptual Model` represents the actual reality which might be interpreted in different ways: two instances are the `Design Model` and the `User Model`. The design model is a `Developer`'s perspective on a design problem. Following their interpretation, they create a `System Image`, which represents a concrete implementation of their tacit knowledge about a certain concept as it is the result of the application of their expertise. From an abstract perspective, a system image may be any kind of artifact; concrete examples are the application introduced in Figure 2.1, its user interface, or the documentation thereof. The system implements the `System Model`, which reflects how the system is actually working, i. e., the outcome of algorithms or the behavior of the user interface. The user model is the `User`'s mental model. While it highly depends on educational, cultural, or other general knowledge models that can be summarized under tacit knowledge, the user model is further affected, i. e., created and evolved, by the system model.

The system image reveals the deviations of the models, in particular between the design model and the user model. Norman proposed to strive for a natural mapping [226], which strives for a minimal amount of cognitive efforts invested by the users, which is achieved by relying on common clues and interaction patterns. Another way of closing the gap between both models is the communication of concepts and ideas, i. e., through the use of scenarios; created by the developers, scenarios describe the indented flow of events with the system image, which helps users to create or evolve their mental model.

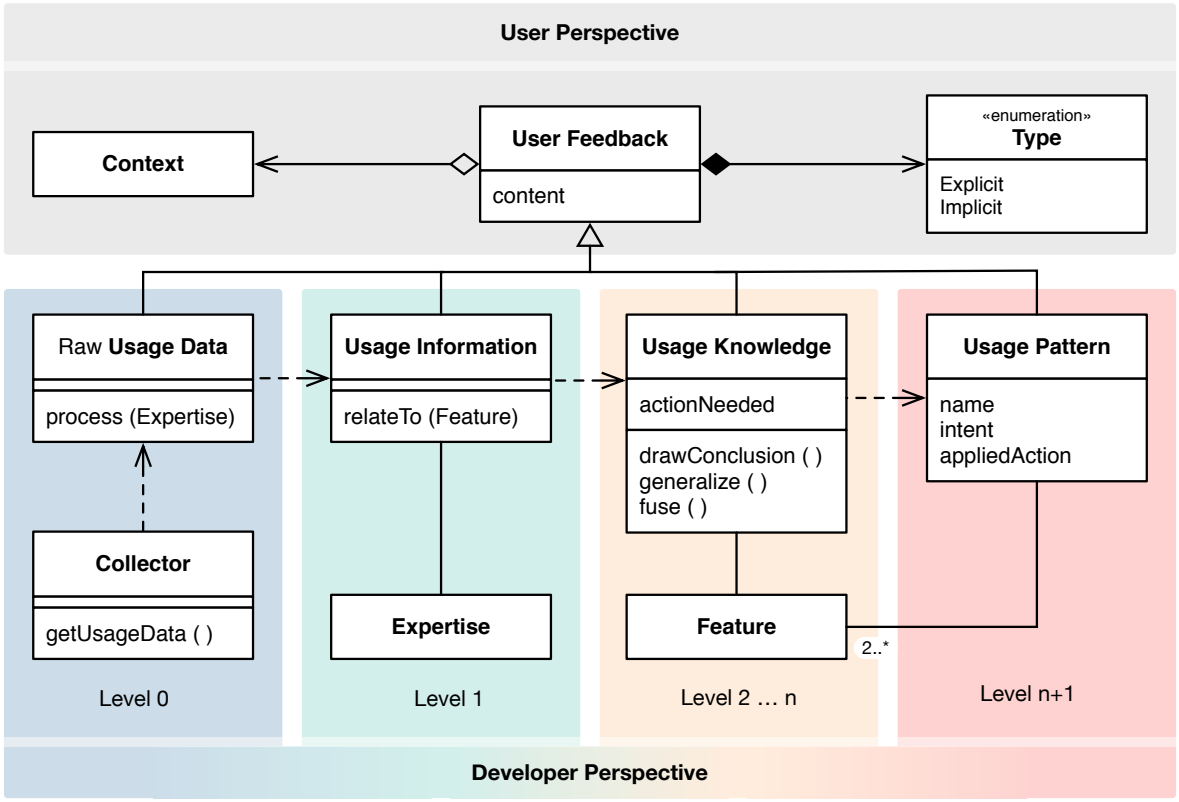


Figure 2.3: A taxonomy of user feedback (UML class diagram).

Finally, the system image allows to derive deviations of the models by observing the users' behavior and comparing it to the expected or the intended behavior. As described, the users' interaction is driven by their tacit knowledge. This may be reflected in accomplishing a task, which allows developers to draw conclusions about interaction model or the need for new features, or bypassing a task, which points toward waiting times, missing interaction steps, or other issues. The feedback of users serves as a means to understand deviations in the conceptual models. The details of the deviations help to extract and utilize the users' tacit knowledge. Therefore, we analyze the structure of user feedback.

2.1.3 User Feedback

We use Figure 2.3 to depict the composition of `User Feedback`, which reflects a user's subjective perspective on an application, or generally any kind of system image. The `content` of the user feedback is the embodiment—i. e., the externalization in a structure that follows the characteristics of the feedback—of the users' mental model (Section 2.1.2) and reflects impressions at a specific point in time.

The user feedback can be of different `Types`: explicit or implicit, as described by Maalej *et al.* [196]: `Explicit` feedback reflects a response that was deliberately initiated by the user with respect to an aspect of an application. Typically, it can

be characterized as a discrete, bounded feedback. `Implicit` feedback reflects a response that results from monitoring the users' behavior or interaction with an application; while the user can be informed about the ongoing monitoring process, they are not actively contributing or driving the feedback, nor do they influence what is currently being recorded. Implicit feedback is characterized by a continuous flow of feedback, without a clearly defined end or beginning. As a result, the user feedback types, i. e., implicit or explicit, further characterize the source of the user feedback. We model `Context` as an additional attribute of user feedback. It enriches the user feedback by providing more insights about it, such as preceding or subsequent interactions. In the human–computer–interaction domain, context is defined as any kind of information that is used to characterize an entity [1, 79]. For user feedback, this may be the time or location of the user.

Developer establish their own perspective on user feedback: While user feedback is closely bound to the users' perspective, the developer's perspective represents an abstraction, which detaches the feedback from the user and tries to process it in an objective manner. We depict this by creating subclasses of user feedback: As user feedback requires any kind of interaction with an application, we add the prefix `Usage` to the child classes to reflect the entities with which the developers interact. We distinguish between different levels of user feedback. This distinction is motivated by the Hearsay project [190], which utilizes a blackboard pattern as described by Buschmann *et al.* [52] in order to incrementally improve a solution space. We further detail the meaning of levels in Section 7.1, in particular the components that transfer user feedback into a higher-level representation.

For the first three levels and for the highest-level of user feedback, we base our model presented in Figure 2.3 on the data, information, knowledge, wisdom hierarchy, which is known as the DIKW pyramid and addressed by many researchers [2, 139, 263]. The terminology of the DIKW pyramid is also adapted in software engineering. For instance, Anquetil *et al.* defines knowledge as *"a net of information based on one's particular experience"* when managing knowledge, while they describe information as data that is enriched with a context [16]. Paech distinguishes *"Data [as] any representation of the world [...]. Information results from data [...], while the transformation of information to knowledge involves intelligence and learning"* [233].

We refer to the first three levels as `Raw Usage Data`, `Usage Information`, and `Usage Knowledge`, which we depict in Figure 2.3. The navigation between them is directional, which means that user feedback can only be transferred into a higher-level representation, but not backwards. Usage data represents the lowest-level representation of user feedback, which is why we add *raw* to its name and refer to it as *Level 0*. It describes the concatenation of a type-specific representation of data. The usage data provided by a `Collector` through `getUsageData` depends on the type of user feedback, as described in the following example.

Example 4: For explicit user feedback, instances of a collector may be any form of a website or an app store dialog. A text field or a star rating bar reflect the means for input that are employed by the collector to gather data from the user: Usage data are the textual representation of the feedback or the number of stars, respectively. For implicit user feedback, a collector is represented by a monitoring component. This can either be a hardware sensor, such as a camera that provides a stream of physical measurements, or a software sensor, such as an analytics framework that observers interactions and events within the graphical user interface. In the first case, usage data could be recordings of the user's face when responding to the application. In the latter case, usage data could be traces that describe users' navigating through the interface or tapping on buttons.

In order to transfer user feedback from usage data to `Usage Information`, it is required to `process` the data using a form of `Expertise`. User feedback in form of usage information provides access to an insight that is valuable to the developer. The processing can be of different complexities; for some usage data, it might be sufficient to perform a basic pre-processing or clean-up step to remove noise. Other usage data may require the application of more advanced expertise that relies on additional domain knowledge. For instance, the usage data could be reallocated or combined with itself to create usage information; in such cases, the expertise can be described as a set of rules or a template which needs to be filled in. Other cases might rely on a black box approach, such as machine learning classifiers, which were trained based on labeled data that shares similarities with the usage data at hand.

Example 5: Following the previous example, the processing of explicit usage data may include the detection of a positive or negative sentiment based on the text provided by the user, or the calculation of an average of multiple star ratings to determine the tendency of appreciation by the users, respectively. The process of deriving usage information from implicit usage data may contain the translation of video data into an annotated observation protocol, or the grouping of interaction sequences and the removal of noise in interaction traces.

While usage information allows developers to detect issues with the application, this user feedback represents a self-contained piece of information which lacks a connection to feature. Therefore, in the context of software development, we define `Usage Knowledge` as higher-level usage information which is enriched by a reference to which the usage information `relatesTo`. We use the `Feature`[5] un-

[5]The notion of a feature depends on multiple aspects [32]. On the one hand, different processes, such as *Requirements Engineering* or *Usability Engineering*, established different understandings, for example with respect to the size of a feature. On the other hand, the term is overloaded, in particular when considering its usage for machine learning models. In this dissertation, we use the notion of a feature for any functional or nonfunctional improvement that can be developed in a single branch. In addition, the feature must apply to the concept presented in Figure 7.4.

der development as a reference point, which eventually connects the user feedback to the developer's development environment. The transitive closure enabled by the feature relationship allows to match the user feedback to the stakeholder of a feature in case `actionisNeeded`. The developer attempts to resolve the issue by `drawingAConclusion`.

> **Example 6:** Let us inspect the following explicit user feedback: *"I recently updated to the latest build of the app and found myself desperately searching for the ability to alter the map layer. It took me a while to understand that there is a new button in the top right corner"*. This extract may be drawn from an app review after version ② of Example 3 had been released. Multiple parts of the extract determine the classification of the user feedback as usage knowledge: While the user's desperation can be clearly interpreted as a negative sentiment that requires the developer's attention, the reference to the latest update, which is further supported by the problem description, allows us to relate the feedback to a particular feature.

As there are multiple data collectors which ultimately lead to multiple instances of usage knowledge that relate to the same feature, it becomes possible to incorporate additional usage knowledge contributed from other usage data collectors. Thereby, usage knowledge can be further refined and improved: the `fuse` method increases the abstraction of the usage knowledge upwards on the level hierarchy.

After applying multiple iterations of usage knowledge fusion, some usage knowledge may qualify to be `generalized` toward a `Usage Pattern`. Usage patterns are generalized from user feedback that is provided for different features, but share similar characteristics—in particular with respect to the `appliedSolution` when addressing the need for action and drawing conclusions that resulted from the usage knowledge. The application of usage patterns must be possible for at least two different, i. e., independent, features. The exploitation of usage patterns is out of scope for this thesis, however, we suggest to follow the definition of Gamma *et al.* [117] for the design of such, i. e., establishing dedicated `names` or `intent`ions which steer the reuse of the usage pattern.

2.1.4 Engineering and Management Processes

The feedback of users and derived needs initiate and fuel the development of applications and improvements. Various processes were established that are concerned with the capture, understanding, and realization thereof. In the following, we summarize requirements engineering, usability engineering, and rationale management. For all processes, we provide a general description followed by its current and future relationship for continuous user understanding.

Requirements Engineering

"Requirements [define] what the stakeholders [...] need from [a system] and also what [it] must do in order to satisfy that need" [80]. Two classes of requirements can be distinguished [45]: functional requirements reflect implementation-independent details that describe the interaction between the system to be developed and a stakeholder, while nonfunctional requirements describe how such interactions shall be achieved. Nonfunctional requirements can be further subdivided, while many proposals on that account are centered around a differentiation into usability, reliability, performance, and supportability [124, 125].

Requirements engineering deals with multiple activities that can be summarized under the discovery, documentation, and maintenance of requirements [292]. The aspect of *engineering* implies that the activities are performed in a structured and well-defined manner [292]. Bruegge and Dutoit describe requirements elicitation as one of two core activities that are performed during requirements engineering [45]: The elicitation of requirements represents the process of deriving a system's specifications that the user understands. It is created by requirement analysts or developers through discussions with users and on the basis of their feedback.

Information on the actual end users and on how they practically employ a software system might not be available at the beginning of the requirement elicitation phase. Therefore, over the last years, in particular implicit user feedback that is collected through usage monitoring gained in relevance for requirements engineering. Data-driven requirements engineering promises to provide insights on users' needs [197], which also raised the demand for new tools to manage complexity that results from the amount of data available for requirements engineering [115].

Usability Engineering

Usability represents a nonfunctional requirement, which classifies usability engineering as an activity that is performed during requirements engineering. Usability is defined as *"the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component"*[6]. Lehman defined a program's usability as one of the main concerns in a changing world [188]. In 2015, Boehm identified eight trends for the future of software and systems engineering processes, one of which is the *"increased emphasis on users and end value"* [37].

While not limited to usability engineering, the elicitation of usability requirements relies on the concept of a stakeholder [63]. Stakeholders are representatives that share the same or different interests in an outcome; Customers, developers, and the users are examples for stakeholders, while often the interests of the latter

[6] IEEE Standard Glossary of Software Engineering Terminology. 1990. IEEE Std 610.12-1990, p. 80, available at https://doi.org/10.1109/IEEESTD.1990.101064.

are neglected as they are difficult to identify in case the user does not actively participate in the requirements elicitation phase [63, 288]. The concept of a persona, i. e., a synthetic representation of a potential end user [64], tries to overcome this issue. In addition, scenarios and use cases help to formally elicit and understand the nonfunctional requirements [45, 56], such as usability requirements.

Nielsen defined usability as a set of multiple attributes that describe different dimensions of a user interface: learnability, efficiency, memorability, errors, and satisfaction [216]. He described usability engineering as the combination of multiple activities that are instantiated during a product lifecycle in order to address its usability [216]. He listed various approaches, such as the application of usability evaluation, i. e., the inspection of the user interface by an expert following well-defined heuristics, or usability testing, i. e., the assessment of application prototypes through interaction by future users [216]. Eventually, he proposed multiple assessment methods that go beyond testing, such as user observing or usage logging [216].

In this regard, usability engineering overlaps with other research areas, such as human-computer interaction, and makes use of a mixture of different techniques, such as experimentation. With respect to continuous user understanding, a recent trend is continuous experimentation, which employs different quantitative and qualitative measurements [17]. For example, Facebook is heavily relying on A/B testing to elicit requirements without a specification upfront [102].

Rationale Management

In the context of software engineering, rationale represents the *"reasoning that lead[..] to [a] system, including its functionality and its implementation"* [45]. This makes rationale management a critical factor for the success of software evolution [45], as the availability of rationale helps to reconstruct decision-making processes [86].

Rationale management serves as the glue between different processes; for instance, Nielsen acknowledged that usability engineering should not be performed in isolation from other, product-related activities [216]. In particular, planning problems require that decision makers consider multiple aspects of a decision and assess them against criteria to find an optimal solution [255]. Therefore, Kunz and Rittel introduce different elements in order to model the rationale for a problem at hand [183]: A simplified version of a rationale model may consist of the rationale elements issue, alternative, pro- and con-argument, as well as a decision.

With the availability of new triggers for decisions knowledge [167], the integration into agile practices [170], additional sources for rationale [7], and new tools in configuration management [168], rationale management evolved, which leads to a better integration into the development process. Notably, the insights derived from continuous user understanding activities could be applied as pro- or con-arguments to support or attack solution proposals or even derive new proposals [153, 158].

2.2 Methodologies

In this section, we describe methodologies that are used throughout the dissertation. While the individual frameworks, models, strategies, and approaches are applied across several domains, the descriptions in the following focus on an introduction for an application in software engineering research.

2.2.1 Design Science and Technical Action Research

Hevner *et al.* describe design science as an *"exten[sion of] the boundaries of human and organizational capabilities by creating new and innovative artifacts"* [142]. Wieringa describes the design science methodology [310] and discusses an extension for action research [311]; on both of which we base the following summary.

The core of design science is the creation of an artifact, for example a tool, a method, technique, or a component, which can be used to achieve a goal. The artifact is applied to a problem context, which may be a composition of software and hardware or a physical environment that includes a human stakeholder. Through interaction with the problem context, the artifact contributes to solve a problem.

Design sciences is driven by two research problems: design problems and knowledge questions. Design problems strive for change, as they result in a treatment that can be applied to a problem context. A treatment reflects a solution, while multiple solutions may exist. Knowledge questions, on the other hand, are concerned with the inspection of the status-quo, i. e., an analysis of the problem context, or with questions that relate to the designed treatment, i. e., its validity to solve a problem.

Both activities, designing an artifact and answering knowledge question, affect each other; they are performed in cycles, i. e., in a design cycle and empirical cycle, respectively. They are embedded in a higher-level process as depicted in Figure 2.4.

The engineering cycle consists of five main tasks: First, during problem investigation, the phenomena of the problem is studied, searching for the reason of a problem. This phase addresses only knowledge questions, which we indicate with the keyword **investigate**, such as participating stakeholders or existing frameworks for the solution of a problem. Second, during treatment design, the researcher is concerned with the creation of a treatment as one solution to the problem at hand. Therefore, they address design problems, indicated by the **design** keyword of the related activities, by specifying requirements and designing the treatment, in case no other treatment has been found. Third, during treatment validation, the researchers use multiple knowledge questions to investigate on how the designed treatment *would* address the problem at hand. This is different from the subsequent phases, the treatment implementation and implementation evaluation, which are performed in an actual problem environment, analyzing the success of the treatment. This serves as the starting point of a new problem investigation.

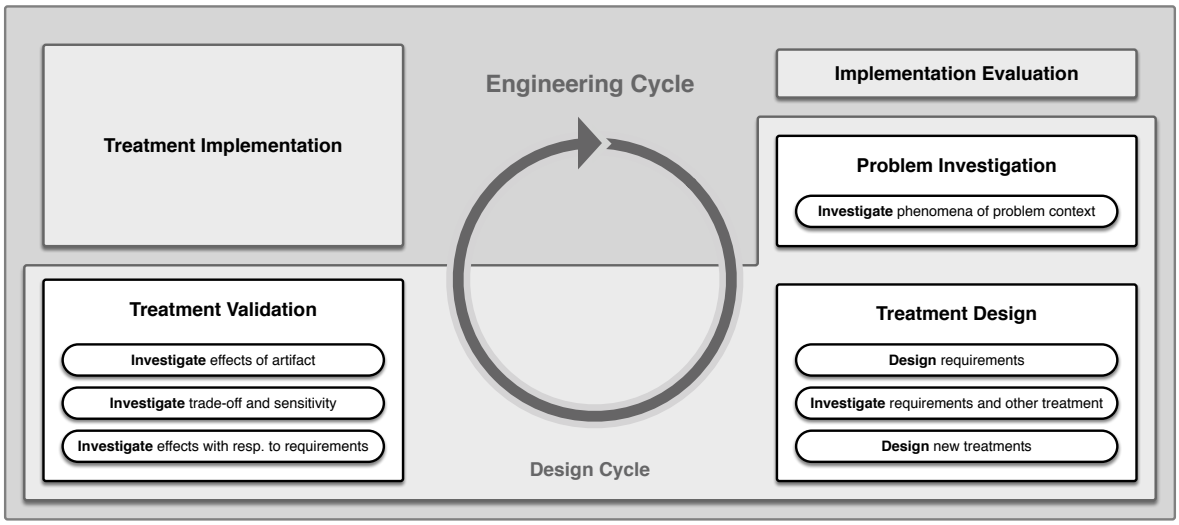


Figure 2.4: The cycles of the design science approach adapted from Wieringa [310].

The first phases, i. e., the problem investigation, the treatment design, and the treatment validation, form a subset of the engineering cycle; they are referred to as the design cycle, which is performed multiple times by the researcher.

Researchers may not be able to perform the implementation and evaluation; for these cases, Wieringa and Morali propose the use of technical action research to achieve to goals [311]: first, to help a client with a problem; second, to learn more about the effects of a treatment in a similar environment. Therefore, the researchers can initiate a so-called client cycle, in which they apply an instance of the treatment that is specifically designed to help the client. The results contribute to a higher-level empirical cycle and help to answer knowledge questions.

2.2.2 Empirical Research Strategies

Easterbrook *et al.* distinguish between five different classes of *methods* that are relevant for software engineering [89]. Runeson *et al.* argue that *ethnographic studies* are a subset of *case studies*, reducing the number to four [265]. We treat *action research* as a higher-level approach in the context of the design science methodology. This leaves us with the following three strategies to empirically gather insights within this dissertation: Case studies, for which we chose to follow the definition of Runeson *et al.* [265]; surveys, following the definition of Fink [106]; and experiments, following the definition of Wohlin *et al.* [314].

All strategies share a similar structure given the basic skeleton of empirical studies that can be captured in five activities [314]: *Designing*, *Preparing*, *Collecting*, *Analyzing*, and *Reporting*. Some of the strategies exhibit subtle difference in strategy-specific wording as well as in the manifestations of their activities: We identify these insights by separating the particular activity into two parts and by introducing three

major phases that allow comparison between the strategies using Figures 2.5, 2.6, and 2.7: *Preparation*, i. e., activities that need to be performed before data can be collected; *conduction*, i. e., activities for or during the execution of an empirical strategy in order to collect quantitative or qualitative data; *utilization*, i. e., activities that are concerned with transforming the data into a higher-level representation after they have been collected.

Empirical strategies are independent of the means of how to collect data, such as through the application of questionnaires or the performance of focus groups. The choice of a means directly impacts the threats to the validity of the preparation, conduction, and utilization of the applied empirical research strategy. In this dissertation, we follow the taxonomy of Runeson *et al.* [265] when reporting threats: *Construct validity*, i. e., whether the gathered observations serve as a means to describe the problem under investigation; *internal validity*, i. e., whether other factors rather than the ones investigated influence the problem under investigation; *external validity*, i. e., whether the derived assumptions can be generalized for reuse with similar problems; *reliability*, i. e., whether another researcher draws the same conclusions provided the same observations.

Case Study

"Case studies investigate phenomena in their real-world settings" [265]; Runeson *et al.* identify the real-life aspect of gathering insights as one of the core characteristics of a case study [265]. While Kitchenham *et al.* use the term observational studies, they also stress the goal of collecting first hand insights from an industrial problem context [166]. This also justifies the need to be precise in formally identifying, defining, and reporting any contextual information that is of relevance for properly understanding the study's observations and results [166]. The focus of investigation is one particular instance of a problem, or a small set of such [265], while in the latter case, they might be interrelated with each other.

Benbasat *et al.* refine the definition of case studies by highlighting the aspect that only a few individuals, which may be represented by a natural person, a group of such, or a representative of an organization, are part of the examination when collecting insights [31]. Runeson *et al.* describe this aspect based on the definition of Robson *et al.* [257]; they stress to draw insights from *"multiple sources of evidence"* [265], which may be in form of one or more individuals as a knowledge source, or multiple means of data collection.

Finally, case studies are characterized by the inability to ensure complete experimental control [265]. This is based on the lack of boundaries of the phenomena under investigation [31]. As a result, case studies tend to be inductive [135], i. e., they aim for the creation of a theory based on a set of observations.

We instantiate the five activities of empirical strategies following the description

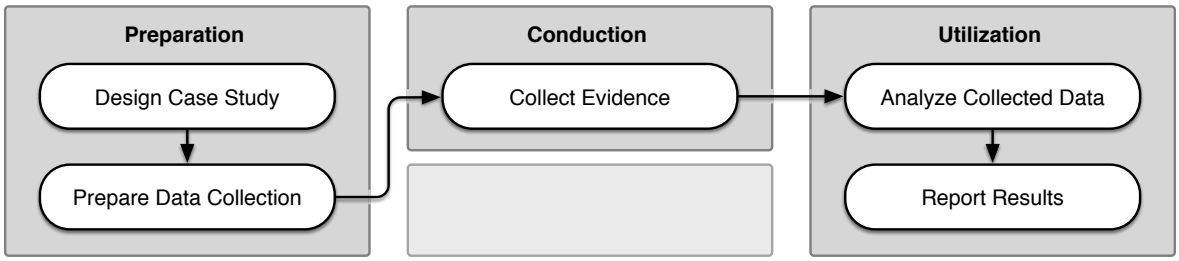


Figure 2.5: Activities to perform a case study based on Runeson *et al.* [265].

of Runeson *et al.* in Figure 2.5. During the *Design [of] the Case Study*, the researcher is concerned with the creation of objectives and the definition of the plan that should be followed to conduct and utilize the case study. These aspects are formalized in the *Prepar[ation of the] Data Collection*, in which specific instruments for the data collection and protocols are determined. The conduction of the case study focuses on the *Collect[ion of] Evidence*, which puts the use of instruments by the researcher into the industrial practices. Hereafter, during the *Analy[sis of] Collected Data*, the researchers apply data analysis methods to extract insights, which they then *Report [in form of] Results*.

Survey

A survey is a systematic procedure with the goal to collect insights from a sample individuals [314]. It is usually performed in retrospective, i. e., after a tool or process has been introduced, in order to understand its effects and performance [245]; this helps to identify emerging needs or segment a population under investigation. *"Surveys aim at the development of generalized conclusions"* [314]—this reflects a key characteristic of the survey research, as it allows to make inferences about a population given only a small sample of individuals [204].

Following Wohlin *et al.*, surveys are conducted on the basis of interviews or questionnaires [314]. Stol and Fitzgerald acknowledge that the terminology is overloaded by stating that *"[t]he term "survey" is equally problematic; in most cases it refers to a sample survey"* [298]. In this dissertation, we follow the intention of Stol and Fitzgerald and understand a survey as a form of consulting a limited group of individuals; in contrast to the case study strategy for which interviews a preferred, we focus on the means of questionnaires for performing surveys.

Following Fink [106], the design of survey studies follows six activities: *setting objective for information collection*, *designing the study*, *preparing a reliable and valid survey instrument*, *administrating the survey*, *managing and analyzing survey data*, and *reporting the results*. We summarize these activities on the basis of more recent literature [107] in the three phases preparation, conduction, and utilization as depicted in Figure 2.6 and describe them in the following from the perspective of a researcher.

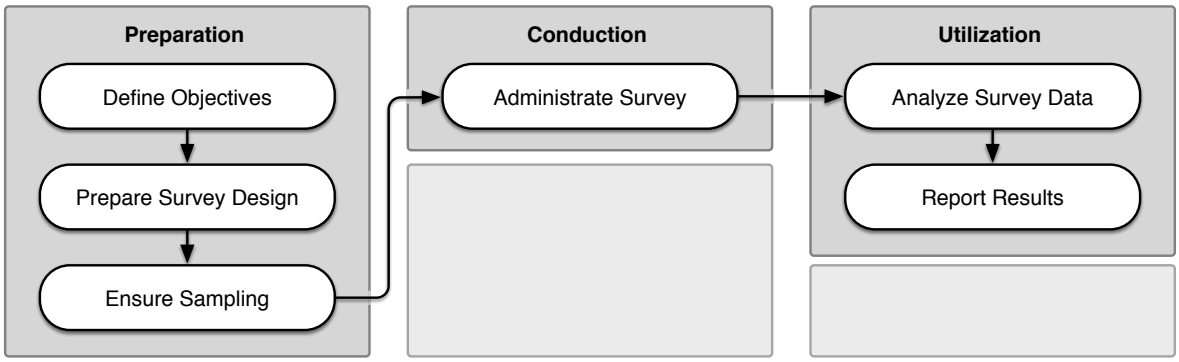


Figure 2.6: Activities to perform a survey based on Fink [106, 107].

The *Define Objectives* activities initiates the design of a survey by describing a goal which should be addressed. During the *Prepar[tion of the] Survey Design*, the researcher defines the environment in which the survey will be performed. In addition, they create the questionnaire which encompasses the content and time constraints. This also includes decisions being made on the medium, e. g., paper-based questionnaires via physical distribution or an online version via email or link. As part of the *Ensur[ance of the] Sampling*, the researcher tries to located the survey in a way it achieves a representative sample. This activity is characteristic to surveys. Hereafter, they *Administrate [the] Survey* which includes the actual execution of the survey with individuals. In the final phase, the utilization, the researcher is concerned with *Analy[zing] Survey Data*, which requires appropriate qualitative or quantitative methods, and the presentation of the results as part of the *Report Results* activity.

Wohlin *et al.* [314] describe the categorization of survey objectives as proposed by Babbie [19]: *descriptive* surveys try to characterize a population at hand; *explanatory* surveys reach out to find reasons and justifications for individuals' behavior and actions; *explorative* surveys typically prepare an additional, more thoroughly conducted investigation and strive to identify major issues of immature solutions.

Experiments

An experiment is a formal investigation of relationships, which is based on a hypothesis or an idea posed by a researcher [28, 314]. An idea is tested by applying a treatment to an object; in most of the cases, this is performed by an individual person, i. e., a subject [314]. An instance of an experiment, a so-called *trail*, is represented by an object that is addressed by a subject through a treatment [314].

Experiments are characterized by *dependent* and *independent* variables: while the first ones are in the focus of an experiment as they are expected to exhibit a change in one or the other way, the independent variables can be controlled by the researchers and are a means to trigger change in the dependent variables [25, 314].

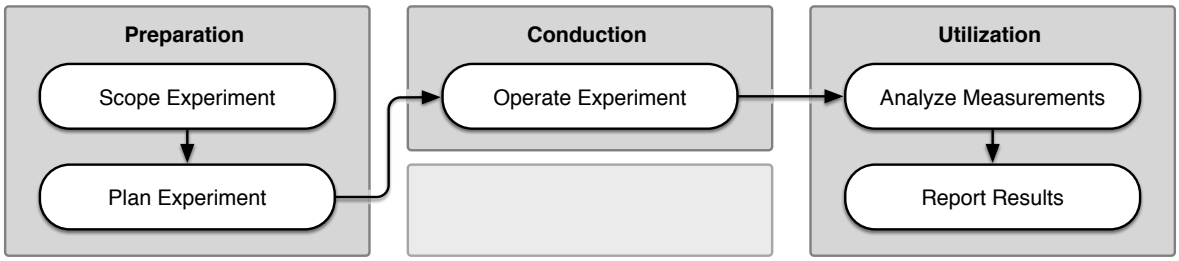


Figure 2.7: Activities to perform an experiment based on Wohlin *et al.* [314].

Runeson *et al.* cite Robson *et al.* [257] to differentiate between a *controlled* experiment that is performed in a laboratory environment and provides a statistical validity with respect to the results [320] and a quasi-experiment: they state that the trail subjects are not drawn randomly from a population in the case of a quasi-experiment [265]. They add that this is the reason why a quasi-experiment and a case study share many common characteristics [265]. In these cases, the results of experiments may not be reproducible [300]. Generally, experiments in computer science appear to be a sparsely used means to evaluate designs or theories [301].

We instantiate the activities for of empirical studies following the description of Wohlin [314] *et al.* in Figure 2.7. The idea for an experiment initiates the *Scop[ing of such]*; this includes the refinement of problems to be studied as well as a goal definition. A detailed analysis of the underlying properties of a dependent variable help to derive better experiment results [69]. As part of the *Plan[ning of] of [the] Experiment*, the means for the conduction phase are selected and threats to the validity of the experiment are evaluated. For the most part, this also addresses the analysis of dependent and independent variables. During the *Operat[ion of the] Experiment*, the researchers are concerned with ensuring a seamless conduction of the experiment trails, while they strive to achieve comparability. As part of the utilization of an experiment, the researchers *Analyze [the] Measurements*, which typically results in quantitative insights. They are described in *Report[ing] Results*, which should also include a description of guidelines and performed steps in order to support others in understanding and assessing the experiment's results [148].

2.2.3 Measurement Strategies

In the following, we present two strategies to measure and interpret data that was collected through the application of an empirical research strategy. The interpretation serves as a means to asses and compare results.

Goal Question Metric Approach

The Goal Question Metric (GQM) approach was proposed by Basili *et al.* as a means for organizations *"to measure in a purposeful way"* [26]. The method originated from

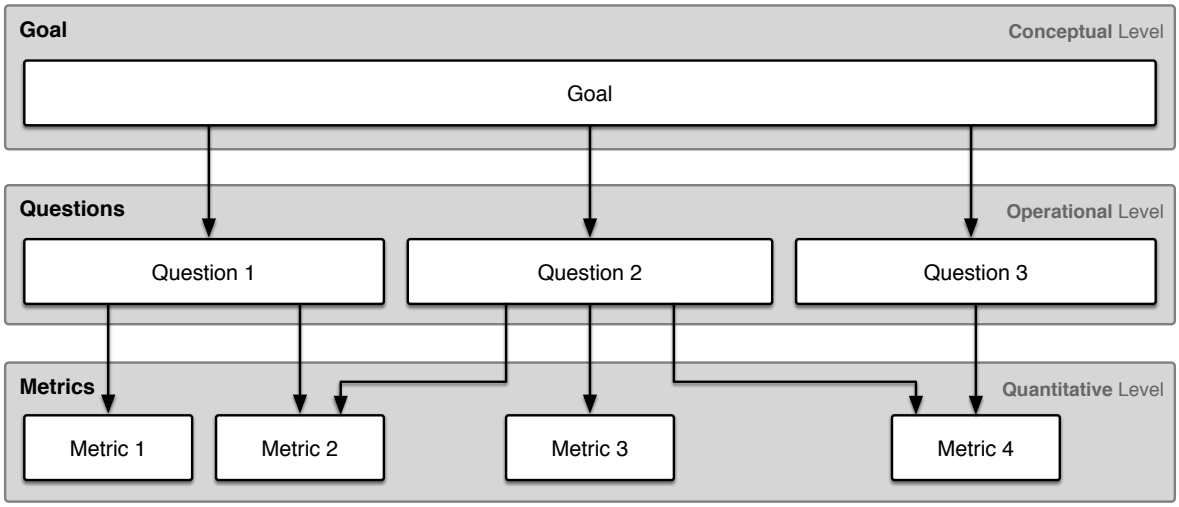


Figure 2.8: A visual interpretation of the Goal Question Metric approach adapted from Basili *et al.* [26].

the attempt to collect goal-direct data [29] and was first presented as part of the TAME project in 1988 by Basili and Rombach [27]. The GQM approach is based on a top-down hierarchy of three layers as depicted in Figure 2.8, which can be summarized as follows [26]. On a conceptual level, a goal needs to be defined in order to provide a measurement for products, processes, or resources. On an operational level, one or more questions are designed in order to characterize the goal from different perspectives. On a quantitative level, one or multiple metrics related to one or multiple questions and serve as a data-driven approach to answer the respective question qualitatively or quantitively.

The result of the specification using the GQM is a measurement model [314]. Multiple suggestions on its application have been made: Briand *et al.* provide a guideline to make the software measurement efficient and useful [42]. Solingen *et al.* provide a theoretical overview which is detailed with practical examples and a series of lessons learned when working with the GQM [305]. Overall, the GQM provides a systematic way to elicit metrics—while related measurements sometimes turn out to be inaccessible to the researcher [265] and thereby further stimulate the investigation of the problem domain.

Technology Acceptance Model

The Technology Acceptance Model (TAM) was originally presented by Fred Davis [73] and refined in the following years [74, 75]: It addresses the need to describe the acceptance of information systems and thereby helps to assess the chances for success of technological solutions. It fulfills the purpose of providing a theoretically justified model by describing a structured procedure to extract unacceptable or appropriative steps as well as explaining behavioral intentions and exploring

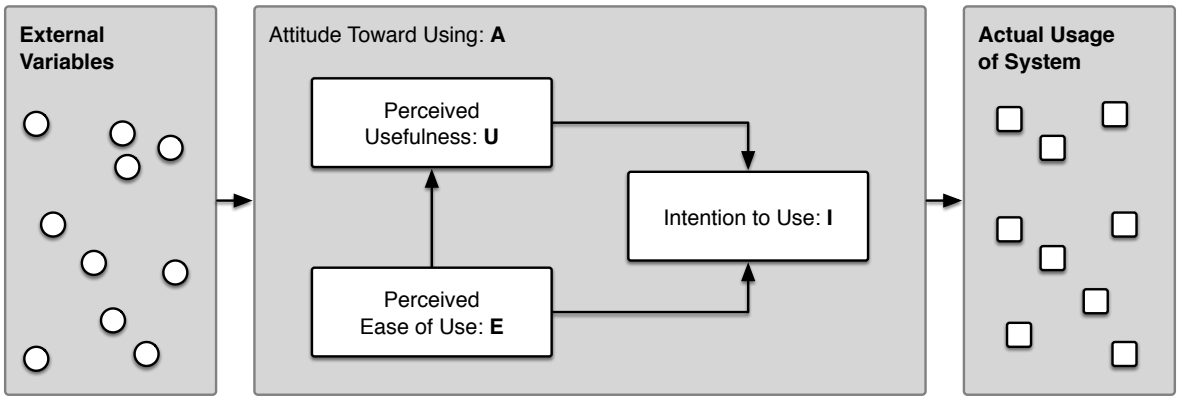


Figure 2.9: A visual interpretation of the Technology Acceptance Model adapted from Davis *et al.* [73, 74, 75].

subjective traces that were triggered by external factors [75]. In Figure 2.9, we provide a visual overview of the core variables of the model and describe them in the following.

According to Davis *et al.* [73, 74, 75], a set of different *External Variables* are affecting a users' *Attitude Toward Using* a system, e. g., their motivation. It determines whether a user will accept or reject a solution at hand and thereby forms the degree of *Actual Usage of [a] System*. In order to further study the users' motivation, their attitude can be separated into two main aspects: *Perceived Usefulness* and *Perceived Ease of Use*, while the latter directly influences the former. The perceived usefulness describes the degree to which an individual would consider a technology beneficial for performing their job, i. e., that it can increase the outcome or improve results. The perceived ease of use describes the individuals' subjective opinion whether *"using a particular system would be free of physical and mental effort"* [73], i. e., how easy they think a technology can be operated. Both aspects ultimately drive the users' *intention to use* a technology. The users' intention is used to predict the actual usage of a system [307]. However, the measurement of actual usage as well its relationship the variables defined in TAM require additional attention following a literature review performed by Marangunić and Granić [201].

The TAM is initially designed for rather mature technologies, as in particular the perceived ease of use of a system requires users to have worked with or to experience a system in order to conceive an opinion. However, Davis and Venkatesh also showed that the model can also serve the purpose of inspecting users' acceptance of a system given only a *preprototype* [76].

Part II
Problem Investigation

PROBLEM INVESTIGATION is concerned with understanding the status quo of a problem context, while we assume that only one answer exists [310]. Therefore, in this part, we detail the problem context and start with a review of literature on the topic of Continuous Software Engineering (CSE) as well as with the introduction of the CURES research project as the basis for the following chapters.

The investigation of the state-of-the-practice is the main contribution of this part. We present the design, i. e., the selection of case and subjects, the data collection procedure, and the analysis procedure, of a case study with 24 practitioners from 17 companies that participated in semi-structured interviews.

We focus on the gain of insights with respect to three knowledge questions: First, how do practitioners apply CSE during software evolution? Second, how do practitioners involve users during CSE? Third, how can usage knowledge support practitioners during CSE? We report the practitioners' answers using qualitative and quantitative measurements and summarize our findings for each question.

Chapter 3

Problem Context Analysis and Research Setup

"While it is true that a great deal of what is generally understood to be logic is concerned with deduction, logic, in the widest sense, refers to something far more general. It is concerned with the form of abstract structures, and is involved the moment we make pictures of reality and then seek to manipulate these pictures so that we may look further into the reality itself. It is the business of logic to invent purely artificial structures of elements and relations. Sometimes one of these structures is close enough to a real situation to be allowed to represent it."

— CHRISTOPHER ALEXANDER [5]

The first half of this chapter lays the foundations for a systematic problem investigation. In Section 3.1, we review literature that describes Continuous Software Engineering (CSE) and derive characteristics separated by categories and elements to define the problem context. On this basis, we present an approach for a systematic integration of usage and decision knowledge into CSE in Section 3.2; the vision originated from a joint research project.

The second half of this chapter initiates the design cycle by summarizing the design of a case study which serves the purpose of addressing the first knowledge goal of this dissertation. Therefore, in Section 3.3, we refine the respective knowledge questions into research questions (RQs) and describe the process of an interview study. Hereafter, in Section 3.4, we elaborate on descriptive study data by detailing the interviewed practitioners, their companies, and projects. Finally, in Section 3.5, we list threats to the validity of the case study and how we mediated their effects.

3.1 Continuous Software Engineering

CSE evolved as a process to develop software on the basis of practices that share frequent and rapid iterations. In this section, we present approaches of describing CSE from which we derive characteristics that are typical for CSE.

Bosch *et al.* [41, 230] introduced the *Stairway to Heaven* that covers four major steps which companies have to take on their transition from practicing traditional software development toward CSE. First, they start with adopting agile practices which lay the ground for the organization of software development. Second, they enable the continuous integration of work, to ensure the product quality after any changes. Third, the continuous deployment of releases enables to gather feedback from users and monitoring the software's performance in the target environment. Fourth, they advance toward research and development as an innovation system—the last and highest-level stage.

In their definition of CSE, Fitzgerald and Stol [109] provide a holistic view on activities from business, development, and operation. They put a special emphasize on the frequent changes in the business environment with which the companies have to cope. Similar to the last step of the *Stairway to Heaven* by Bosch *et al.* [41, 230], they state that CSE is more than the company's adoption of continuous delivery and continuous deployment. According to their descriptions, CSE goes beyond the *DevOps* approach, which describes a state in which software development and its operational deployment are closely integrated. They coined the term *BizDevOps* as a synonym for CSE to emphasize the need to improve the link between business strategy and software development. They define several *"continuous star"* practices, which make up an agenda for CSE [109].

We examined the *Stairway to Heaven* by Bosch *et al.* [41, 230] and inspected the work by Fitzgerald and Stol [109]. Based on their descriptions, we acquired a preliminary list of characteristics that are typical for CSE. This list serves as our starting point of a definition for CSE. We introduce a simple hierarchy with CSE *categories* on the top which are refined by CSE *elements*. We provide a visual overview in Table 3.1.

The frequent involvement of users is a major concept in CSE. Thus, we introduce *user* as a CSE category that refers to both customers who commissioned a project and end-users, as well as the ability to learn from different types of user feedback, such as implicit or explicit feedback. *Software management* includes practices concerning the overall software process. The *development* category is composed of more specific development activities, such as requirements engineering and design excluding implementation and quality assurance. The *code* category includes implementation-related practices, such as *version control* and *branching strategies*. We bundled activities such as *audits* and *pull requests* in the *quality* category. The *knowledge* category collects practices supporting the overall knowledge management.

Table 3.1: Categorization of CSE into categories and elements on the basis of Bosch *et al.* [41, 230] and Fitzgerald and Stol [109]. Adopted from Johanssen *et al.* [157, 158].

CSE Categories	**CSE Elements**
User	Involved users and other stakeholders; learning from usage data and feedback; proactive customers
Software Management	Agile practices; short development sprints; continuous integration of work; continuous delivery; continuous deployment of releases
Development	Continuous planning activities; continuous requirements engineering; focus on features; modularized architecture and design; fast realization of changes
Code	Version control; branching strategies; fast commit of code; code reviews; code coverage
Quality	Automated tests; regular builds; pull requests; audits; run-time adaption
Knowledge	Sharing knowledge; continuous learning; capturing decisions and rationale

We acknowledge that the allocation of multiple elements to their counterpart of a categories is not always straightforward and some allocations may be a matter of interpretation. For instance, we consider arguably technical practices, such as continuous delivery, under the category of software management. We made this decision to highlight their impact on the overall CSE process. Similarly, we include code reviews in the code category to emphasize their operational character, while pull requests, i. e., merging code into another branch or code basis, are understood as quality-related tasks. We refine the list of CSE elements and CSE categories based on interview observations and discuss the issue of ambiguities in Section 4.2.

3.2 CURES Project

We see CSE as an opportunity to support stakeholders in capturing, exploiting, and understanding knowledge [153]. In particular usage and decision knowledge are essential for software evolution and contribute to an improved product quality. Beside their individual benefits, we expect synergies in their combination: for instance, decisions might rely on results that are derived from user feedback. This section describes a vision for integrating usage and decision knowledge into CSE that was developed as part of the research project CURES ("Continuous Usage- and Rationale-based Evolution Decision Support")[7].

[7]The CURES project is part of the Priority Programme 1593 of the DFG, the German Research Foundation. More information is available on the SPP1593 website: http://www.dfg-spp1593.de.

Knowledge Types

Short feedback cycles in CSE enable the direct involvement of users since they have access to the most recent version of the software system. Thus, derived usage knowledge provides insights into the users' acceptance of a software increment and reveals their needs. Usage knowledge helps to improve the usability of a software system, however, user-centered design techniques are *"[...] difficult to master, and [..] inaccessible to common developers and small and medium-sized software development teams"* [284]. With its lightweight and agile character, CSE provides the opportunity to close this gap between software engineering and usability practices.

Developers require rationale to make decisions that contribute to the quality of a software system. Decision knowledge consists of decisions, the issues they address, proposals for a solution, as well as pro- and con-arguments, as described in Section 2.1.4.3. The lack of decision knowledge fosters the erosion of the software architecture or the introduction of new quality problems. Capturing decision knowledge has multiple benefits, e. g., it can improve the decision-making through making criteria explicit and preventing knowledge vaporization [86]. In practice, developers do not capture decision knowledge often enough [6]; CSE offers new opportunities to overcome this problem as it provides multiple practices and documentation locations in which developers can capture decision knowledge [167, 168]. For instance, developers capture decisions when they commit code in commit messages [47].

Vision

Following the benefits of usage and decision knowledge for CSE, we developed a vision of how these knowledge types can be efficiently and effectively integrated. In Figure 3.1, we outline the major components of the CURES vision and describe them in the following in more detail. In general, the vision proposes an extension to a CSE *Infrastructure* which enables the systematic management of usage and decision knowledge during CSE [153].

The CSE infrastructure incorporates the CSE elements described in Section 3.1 and refined in Section 4.2—except for the elements of the user and knowledge categories. The two main stakeholders in the CURES vision are *Developer*s and *User*s. An event-based environment allows the developers to create new releases of software increments and to deliver them to the users at any time [175, 176]. Developers organize their work following a well-defined strategy: they utilize feature branches to create product increments in form of code commits. As soon as they have completed a feature under development, they merge back a feature branch into a master branch. The master branch contains the final software product. Each feature branch on its own can be released to users, allowing the delivery of different proposals for one feature at the same time.

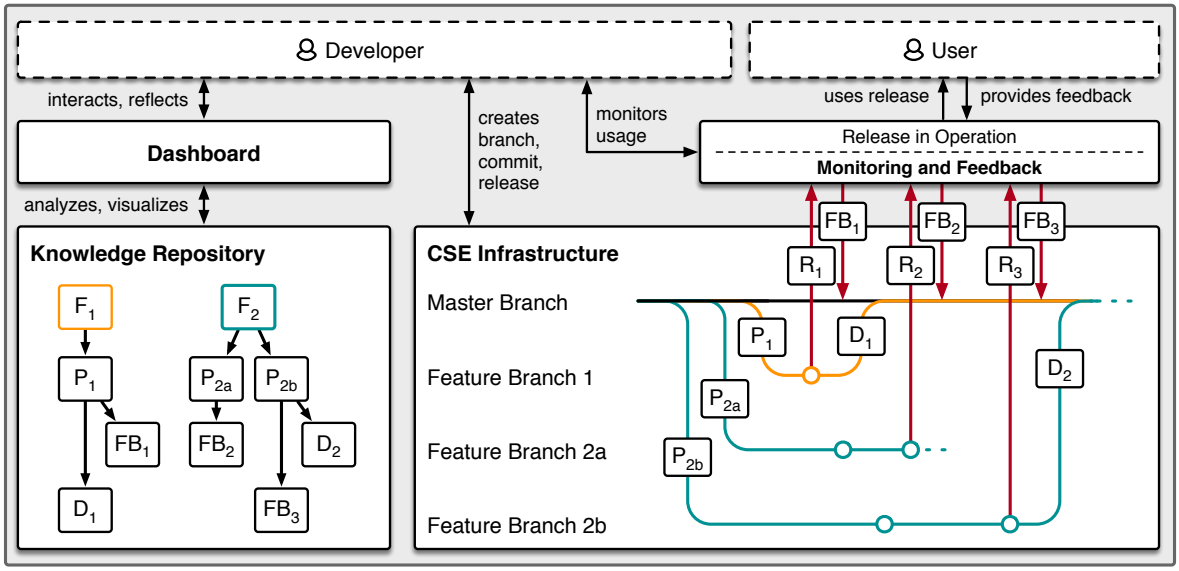


Figure 3.1: The CURES vision as presented by Johanssen *et al.* [153, 158].

A proposal P forms the basis to start work on a feature F. For instance, in Figure 3.1, the proposal P_1 initiates work on the feature F_1. After the first commit, the current state of the feature branch is delivered through a release R_1 to users, who are able to use the latest feature changes in their target environment. A *Monitoring and Feedback* component collects information about the release in operation and offers the possibility to the user to give explicit feedback FB. In the example described in Figure 3.1, FB_1 is provided by the user. The developer makes decision D_1 to merge the feature into the master branch, closing their work on the feature.

The core component of the CURES vision is represented by the *Knowledge Repository*. The knowledge repository continuously stores and relates information collected from the CSE infrastructure as well as from the monitoring and feedback component. A *Dashboard* component enables to access the stored knowledge. For instance, the claimed solution to an issue could be compared to an alternative solution based on the impact of a feature branch in the context of previous decisions. Developers use the dashboard to interact and reflect with the knowledge in order to improve their decision-making and development process in the CSE infrastructure.

The CURES vision supports more complex scenarios. For example, in Figure 3.1, two proposals—P_{2a} and P_{2b}—can simultaneously lead to the development of the feature F_2 in two different ways. Based on the retrieved feedback FB_2 and FB_3, the developer puts the development of F_{2a} on hold and makes the decision D_2 which leads to the merge of F_2 into the master branch. By utilizing the dashboard, the knowledge repository, as well as the monitoring and feedback component, the CURES vision extends the CSE infrastructure with support for usage and decision knowledge. It supports the evolution of knowledge and handles it similar to the evolution of code in a CSE infrastructure.

Challenges

In particular the integration of usage knowledge in CSE arises challenges. In the following, we describe high-level issues that we identified following an analytical analysis of the CURES vision presented in Figure 3.1. To achieve a fine-grained understanding of the challenges and address the knowledge questions, we performed a case study which we introduce in the subsequent section.

Software is characterized by a heterogeneous set of users. In CSE, the user role might be filled by the developers themselves, the customers who initiate the development of an application as part of a contract, or the actual end users of an application. All of them have different intentions, expectations, and approaches toward using an application which needs to be considered.

User feedback can either be explicit or implicit as described in Section 2.1.3. Explicit feedback is actively provided by the user, such as an app store review, while implicit feedback relates to anything users reveal through their interaction, such as pressing a button. It has yet to be answered how both feedback types can be incorporated within the CSE infrastructure: different strategies for feedback elicitation need to be evaluated, such as in-application feedback requests for new features or automatic usage data collection for each feature branch. Enriching users' explicit with implicit feedback in the monitoring and feedback component enables the creation of a comprehensive user understanding. A main challenge is to derive a user's motivation and construct a user behavior model. Additional context information, such as a user's location or current activity, might be useful for creating such a model.

Further challenges arise in the synchronization of releases, feedback, and their granularity: in a CSE infrastructure new increments are released frequently, while usage patterns develop only over time and the difference between releases can be as little as a commit.

3.3 Case Study Design

We relied on the means of semi-structured interviews [213, 265] and performed a case study following the activities described in Section 2.2.2.1 to investigate on the phenomena of CSE in practices. We focused on three areas of interest: current practices in CSE, user feedback in CSE, and usage knowledge in CSE. To report on the case study, we follow the structure proposed by Runeson *et al.* [265]. First, we describe the objectives of the study by refining knowledge questions into research questions in Section 3.3.1. Second, we outline case and subject selection in Section 3.3.2. Third, we describe the data collection procedure in Section 3.3.3. Fourth, we present the analysis procedure in Section 3.3.4. Two researchers were equally involved in each of case study's phases.

3.3.1 Refinement of Knowledge Questions

One of the two knowledge goals of this dissertation is to understand the developers' engagement and demand in the problem context, i. e., during CSE. In Section 1.4.2, we described how this Knowledge Goal 1 can be refined into three knowledge questions. This section describes a further refinement of knowledge questions into research questions which serve as objectives for the case study.

How do practitioners apply CSE during software evolution?

With respect to Knowledge Question 1, the interview study intends to understand how practitioners apply CSE during software evolution. From this goal, we derived the following four research questions that refer to the collection of CSE elements and categories introduced in Section 3.1.

First, we intend to learn about the practitioners' perception of CSE. Further, we want to know whether practitioners define a threshold that needs to be passed before a company can claim to practice CSE. We summarize this as follows:

> **Research Question 1:** How do practitioners define continuous software engineering?

Second, to understand the perception of CSE in more detail, we are interested in the three CSE elements that are most relevant to the practitioners. In addition, we collected applied tools. We summarize this as follows:

> **Research Question 2:** Which elements of continuous software engineering are perceived as most relevant by practitioners?

Third, we want to reveal positive, neutral, and negative experiences with CSE elements. This is of interest for the practitioners who are about to transition toward CSE and lack insights on where to start. We summarize this as follows:

> **Research Question 3:** What are practitioners' experiences with continuous software engineering?

Fourth, we are interested in practitioners' plans for additions in the short and long term. Insights from this RQ help to understand trends of future CSE elements adoption. We summarize this as follows:

> **Research Question 4:** What are practitioners' future plans for continuous software engineering?

How do practitioners involve users during continuous software engineering?

With respect to Knowledge Question 2, the interview study intends to understand how practitioners involve users during software evolution. From this goal, we derived three RQs and nine sub-RQs. In the following, we summarize their motivation using examples or literature when required.

First, we want to understand general aspects of user feedback that affect how user feedback is both captured and utilized. We summarize this as follows:

> **Research Question 5:** Which user feedback do practitioners consider for continuous software engineering?

Therefore, we study the types of user feedback considered by practitioners. User feedback can be explicit or implicit [196]. As described in Section 2.1.3, explicit user feedback refers to any feedback the user is actively providing while being aware of it, such as an application review or answering a pop-up for a star ranking. Implicit user feedback is produced by users interacting with the artifact, such as a click on a button, navigating through the application, or using a certain feature. While each user feedback type is interesting on its own, the combination can be also useful and beneficial. We summarize this as follows.

> **Sub-RQ 5.1:** What types of user feedback do practitioners collect?

Another aspect are artifacts to which practitioners relate user feedback. This can be either to the entire application or to a specific feature. The difference in this granularity indicates the maturity of the user feedback capture process, because feature-specific user feedback requires more efforts in processing. The answers also allow us to assess their model of how they represent user feedback, which should be adjusted to practitioners' needs. We summarize this as follows:

> **Sub-RQ 5.2:** Which artifacts do practitioners relate user feedback to?

Second, we want to investigate how practitioners capture user feedback in CSE. This addresses aspects such as how often practitioners capture user feedback, which tools they use, the sources on which they rely, and whether they capture the context of user feedback. We summarize this as follows:

> **Research Question 6:** How do practitioners *capture* user feedback in Continuous Software Engineering?

The frequency with which practitioners capture user feedback indicates if practitioners also apply a continuous approach following CSE. If not, this would indicate that user feedback is treated differently than other artifacts in CSE and thus indicate unused potentials. We summarize this as follows:

> **Sub-RQ 6.1:** How often do practitioners capture user feedback?

The capture of user feedback results in large amounts of unstructured data that needs to be processed to derive useful insights. This can be done either manually or with tool support, which is either designed to capture user feedback in the first place or represented by standard software. We summarize this as follows:

> **Sub-RQ 6.2:** Do practitioners rely on tool support to capture user feedback?

While we assume that practitioners rely on user- or developer-triggered explicit feedback or implicit feedback from monitored usage data, we also intend to reveal more about the feedback provider, which may be a group of individuals. Therefore, we are interested in practitioners' sources of user feedback, i. e., from whom they capture user feedback. This could be reflected in either a one-way interaction, i. e., feedback from the user without reaction from developer side (e. g., an app store review), or a two-way interaction, i. e., discussion-like exchange of information between developer and user (e. g., an email conversation) or even a fellow developer. We summarize this as follows:

> **Sub-RQ 6.3:** What are practitioners' sources for user feedback?

We intend to understand the capture of the context for the user feedback. The context of user feedback may be anything that is not part of the feedback itself, but helps to generate a better understanding, such as an activity a user was performing while they are using the application. The practitioners' answers might help to understand the scope and relevance of context for user feedback capturing. The responses to this question will allow us to make assumptions about the company's progress in user involvement, since we consider the incorporation of contextual information into the user feedback analysis as an advanced approach. We summarize this as follows:

> **Sub-RQ 6.4:** Do practitioners capture the context of user feedback?

Third, we want to investigate how practitioners utilize user feedback in CSE. Therefore, we address reasons why practitioners capture user feedback, whether they exploit changes over time, and whether they combine different user feedback types. We summarize this as follows:

> **Research Question 7:** How do practitioners *utilize* user feedback in Continuous Software Engineering?

We are interested why practitioners capture the user feedback and what it is used for with respect to the artifact. For instance, it can be used to evaluate the overall

app performance. Also, it can be utilized to evaluate specific features or different implementations. We summarize this as follows:

> **Sub-RQ 7.1:** Why do practitioners capture user feedback?

Investigating the evolution of user feedback can support the practitioners in revealing trends in how feature increments are applied and perceived by users. We expect that answers regarding user feedback over time will contain descriptions of tools and approaches they apply. We summarize this as follows:

> **Sub-RQ 7.2:** Do practitioners exploit the change of user feedback over time?

We are interested whether practitioners combine two or more types of user feedback to derive new insights. An example may be a written review, i. e., explicit feedback, that is augmented with a usage log, i. e., implicit feedback, from the same user. Combining user feedback types requires advanced skills in collecting and assessing usage data. This aspect relates to one of the major goals of CURES vision as described in Section 3.2. We summarize this as follows:

> **Sub-RQ 7.3:** Do practitioners combine different user feedback types?

How can usage knowledge support practitioners during CSE?

As described in Section 3.2, usage and decision knowledge are suitable candidates to be included in the CSE processes to improve the product increments. We were interested in the practitioners' thoughts of the CURES vision. With respect to Knowledge Question 3, the interview study intends to understand how usage knowledge can support practitioners during CSE. From this goal, we derived five RQs.

First, we are interested in the practitioners' overall impression and their opinion on the feasibility of the CURES vision. We summarize this as follows:

> **Research Question 8:** What is the attitude of practitioners toward the CURES vision?

Second, we intend to find the core benefits of the integration of usage and decision knowledge into CSE. Practitioners' responses regarding this aspect strengthen the goal of the CURES vision. We summarize this as follows:

> **Research Question 9:** What are benefits of the CURES vision perceived by practitioners?

Third, by asking the practitioners for major obstacles of the CURES vision, we want to detail the feasibility aspects of the proposed vision. Responses help to understand practitioners' problems. We summarize this as follows:

> **Research Question 10:** What are obstacles of the CURES vision perceived by practitioners?

Fourth, based on the practitioners' perceived benefits and obstacles, we want to ask them to provide ideas for short-term extensions to the CURES vision that do not require major changes. We summarize this as follows:

> **Research Question 11:** What are important extensions to the CURES vision according to practitioners?

Fifth, following the long-term visions of the practitioners, we strive for feature requests that could improve the CURES vision. We assume that they require significant changes to the overall idea. In comparison to RQ 4, this research question has a focus on knowledge during CSE. Furthermore, it requires the practitioners to relate their answers to the CURES vision. We summarize this as follows:

> **Research Question 12:** What are potential additions to the CURES vision according to practitioners?

3.3.2 Case and Subject Selection

We chose a semi-structured interview study as the means for the case study as we focus on knowledge that resides in the minds of practitioners rather than in documents [265]. Also, interviews allow us to study phenomena, actors, and their behavior in their natural context [298] and enable us to clarify problems right away. This contributes to collect more information from the practitioners.

We prepared a questionnaire containing descriptive data questions, of which we report the results in Section 3.4. To address the presented research questions, we created a list of interview questions, which encompasses multiple open questions that included sub-questions to further stimulate verbose answers by the practitioners. The interview questions follow either the RQ or the sub-RQ presented in Section 3.3.1; only slight modifications were made. Therefore, with respect to reproduction of this study, we advise to use the sub-RQs as interview questions to maintain comparability with our results. Figure 3.2 states example questions. We actively encouraged the interviewees to give detailed answers. We planned 90 minutes for the interview, which included research questions that are not further addressed in this article; they are, however, presented by Kleebaum *et al.* [169].

We assembled a list of companies who to our knowledge apply the majority of our preliminary collection of CSE elements (Table 3.1). We formulated a template request mail for scheduling an interview; it is available in Appendix B.1. We attached a slide deck that, among others, contained the following information:

- an introduction to the problem context, i. e., CSE and knowledge types (see Appendix B.2);
- a mindmap with CSE elements and categories (see Appendix B.3);
- a consent to participate in the study (see Appendix B.4).

We asked the interview partners to agree to the interview only if they have worked in at least one project that applied the majority of the CSE elements. We did not restrict interview partners by role descriptions. However, we provided examples of roles that we preferably address, e. g., developer or project manager.

Since two researchers conducted the interviews simultaneously, we set up an interview guideline to ensure comparability. Besides the interview questions, the guideline comprised remarks to increase the questions' understandability. In particular, interviewees were asked to use concrete experiences from one specific project. All interview material was available in English and German. We ran two dry runs with colleagues who have industry experience to practice the interview procedure.

3.3.3 Data Collection Procedure

We sent the interview requests to 22 companies and received 18 replies. One company declined to participate in the study due to secrecy concerns. The researchers conducted 19 interviews between April and June 2017 and one additional in September 2017. In four interviews, two interviewees participated. The interviews were conducted in person or via phone. Descriptive data is provided in Section 3.4. The interviews took 70 minutes on average and were audio-recorded with the permission of the interviewees. We transcribed the audio recordings and sent the transcripts to the interviewees to correct misunderstandings.

In some cases, interviewees provided us with notably relevant feedback after the interview, for example on the go or after some time via email. We collected and added this information to the interview protocol to treat it like the rest of the interview, i. e., by systematically allocating answers to research questions and coding the answers, as described in the following Section 3.3.4. In all cases, we guaranteed the anonymity of the practitioners by only publishing aggregated results.

3.3.4 Analysis Procedure

Two researchers analyzed the transcripts as suggested by Saldaña [268]. We utilized a *qualitative data analysis* software to apply two stages. During the first stage, we allocated answers to a research question as introduced in Section 3.3.1. On one hand, this is necessary as the transcripts are a continuous text. On the other hand, there were cases in which a practitioner came back to a question later during the interview.

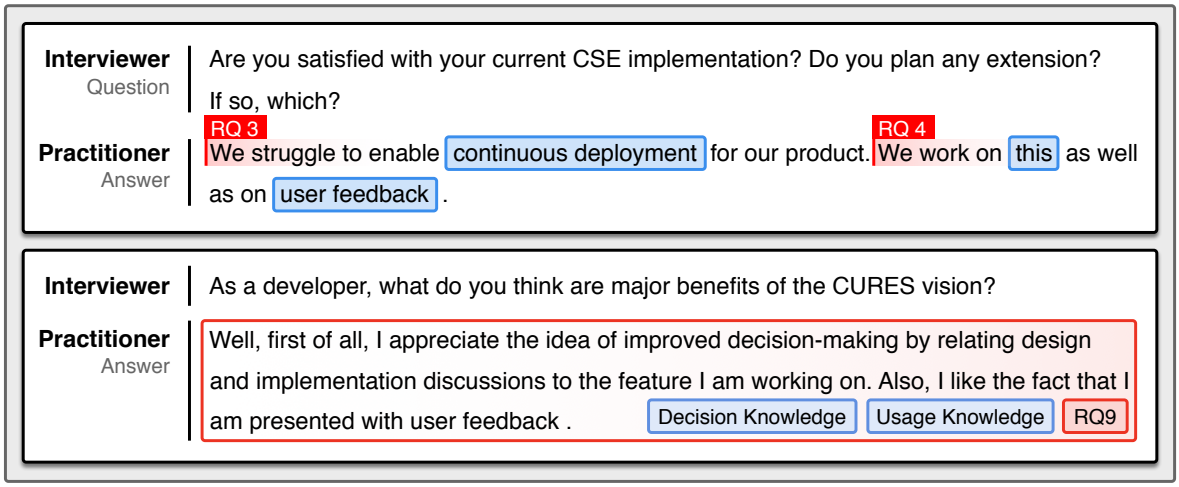


Figure 3.2: Two interview extracts with RQ allocations in red color and applied codings in blue color. Adapted from Johanssen *et al.* [157, 158].

Hereafter, we performed a fine-grained coding stage. In Figure 3.2 we provide two example extracts. The upper part shows allocations of answers to RQs highlighted in red, while these may contain multiple occurrences of fine-grained codes, shown in blue. The lower part shows an example in which one sentence could relate to multiple RQs and codes.

For the first stage, i. e., allocating answers to RQs, two researchers analyzed a single interview to measure the intercoder reliability. The representativeness and completeness were criteria for choosing the interview for this procedure. One researcher found 85 answers related to the RQs, the other 77. Given the total number of 162 instances, the researchers matched in 134 and mismatched in 28, leaving a result of 82.72 % in equally allocated answers. The 28 mismatched instances were jointly discussed and resolved by mutual consent. The discussion necessary for this purpose strengthened the shared understanding of the researchers. We observed that almost all mismatches were caused by a missing allocation, not by the allocation to different RQs. Therefore, in case of doubt, we agreed on allocating multiple RQs to an answer to prevent information loss.

The allocation to research questions was always made at sentence-level, i. e., included one or more sentences. After the allocation of answers to the research questions that addressed Knowledge Question 1, we updated our initially elected list of CSE elements in Table 3.1 on the basis of the insights we derived from the answers. The updated collection is presented in Section 4.2, along with the rationale for the applied changes. After the allocation of answers to the research questions that addressed Knowledge Question 2, we decided to update ten of these allocations due to a more sharpened understanding of minor differences, such as regarding the short-term extensions (RQ 11) and long-term additions (RQ 12), which sometimes appeared to be similar and only differed in minor nuances, or were actual bene-

Table 3.2: List of codings applied to sub-RQs for Knowledge Question 3.

RQ	Summary	Coding
5.1	Type of User Feedback	Explicit; implicit.
5.2	Artifacts for User Feedback Relationship	Application; feature.
6.1	Frequency of User Feedback Capture	Event-based; periodic; continuous.
6.2	Tool Support for User Feedback Capture	Tool support; manual capture.
6.3	Sources for User Feedback	Internal; external.
6.4	Context of User Feedback	Capture context; omit context.
7.1	Reasons for User Feedback Capture	Planning; support; improvement.
7.2	Change of User Feedback over Time	Exploitation of change over time; no exploitation.
7.3	Combination of User Feedback Types	Pragmatic combination; no combination.

fits (RQ 9) overseen by the practitioners. With respect to Knowledge Question 3, we allocated the answers to the respective sub-RQs to provide a more fine-grained starting point for the subsequent phase.

The second stage was comprised of adding codings to these answers. For RQs that addressed Knowledge Question 1, we used the updated collection of CSE elements as fine-grained codes at word-level. For sub-RQs that addressed Knowledge Question 2, we identified emerging topics that we listed in Table 3.2 and coded the answers in terms of these topics. Answers to some sub-RQs qualified for more than one coding. For RQs that addressed Knowledge Question 3, we coded the answers in terms of their focus on a sentence-level. We distinguished between a *usage knowledge focus*, *decision knowledge focus*, or *other focus*.

The coding was done manually, since searching for keywords is insufficient, provided that practitioners use varying formulations to describe the same aspect. For instance, in Figure 3.2, *this* refers to the CSE element *continuous deployment*. Occasionally, we had to decide which code to use based on the context. We practiced the coding and agreed to prefer to code an instance if in doubt. Each researcher coded the interviews they conducted on their own.

We analyzed the results both quantitatively and qualitatively. In case we coded a practitioner's answer to a research question more than once with the same code, we recorded only a single occurrence to receive binary results. We analyzed the interviews on the interview-level, and not on person-level, which means that—in case two interviewees participated in an interview—their answers were treated as one subject. With respect to Knowledge Question 1, i. e., how practitioners apply CSE during CSE, we summarize the results in form of observations (Section 4.1) and derive a model for CSE (Section 4.2). With respect to Knowledge Question 2,

i. e., how practitioners involve users during CSE, we quantitatively assess their answers (Section 5.1) and provide five recommendations (Section 5.2). With respect to Knowledge Question 3, we summarized the results in form of groups (Section 6.1); in this dissertation, we focus on the code *usage knowledge* and present a revised version of the CURES vision (Section 6.2). Answers from at least two individual interviews were required to form an observation or a group; for Knowledge Question 3, this may include knowledge codes that are not further detailed in this dissertation. Also, answers from a single interview can account for more than one response per topic, while one response can relate to more than one knowledge focus. An initial version of this paper was sent to the interviewees to validate the interview results.

3.4 Descriptive Case Study Data

We report on descriptive data about the companies, practitioners, and projects that were analyzed. Overall, we interviewed 24 practitioners from 17 companies during 20 interviews. One company was interviewed twice; another one three times. We aimed for a high diversity of interviews by approaching companies of different size, practitioners with different backgrounds, and projects in different domains. In each interview, the practitioners related their answers to one particular project; four interviews were attended by two practitioners.

Companies

We consider four companies (24 %) as small and medium-sized enterprises[8] (SME), which means a maximum staff headcount of 250. We refer to the remaining 13 companies as corporations; eight employ up to 2,000, two around 50,000, and three 100,000 or more employees. We report the overall number of employees, since we assume that the CSE process is not limited to a specific role from the development team. Figure 3.3 visualizes the practitioners' answers regarding the company.

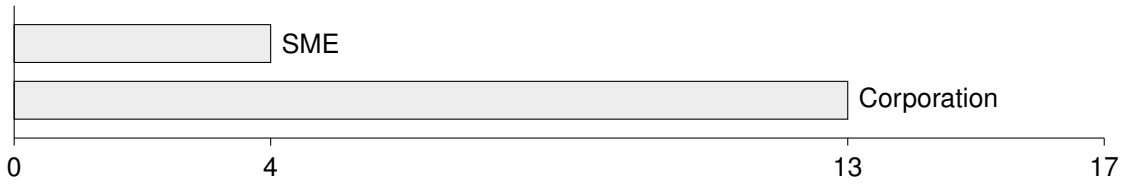


Figure 3.3: Number of companies distinguished by their size.

Seven (41 %) of the interviewed companies offer consultancy services, mostly to other businesses, while ten (59 %) companies develop software products for the consumer and business markets.

[8] As noted at http://ec.europa.eu/growth/smes/business-friendly-environment/sme-definition.

Practitioners

Based on their role description, we grouped the 24 practitioners into five categories: *developers* (25 %), e. g., a software engineer, *project managers* (25 %), a role with project-focused responsibilities, and *technical leaders* (25 %), a role with technical-focused responsibilities; CSE *specialists* (21 %), a role with a CSE reference, e. g., a continuous deployment manager or a DevOps engineer, one practitioner reported as an *executive director*. On average, the practitioners have spent two years in the respective role. All practitioners hold a Bachelor or Master degree with three-quarters in a field in or close to computer science; 16% hold a PhD. With one lacking response, on average, 23 practitioners have 10 years' experience in information technology (IT) projects and participated in 19 IT projects. Figure 3.4 visualizes the practitioners' answers regarding their role description.

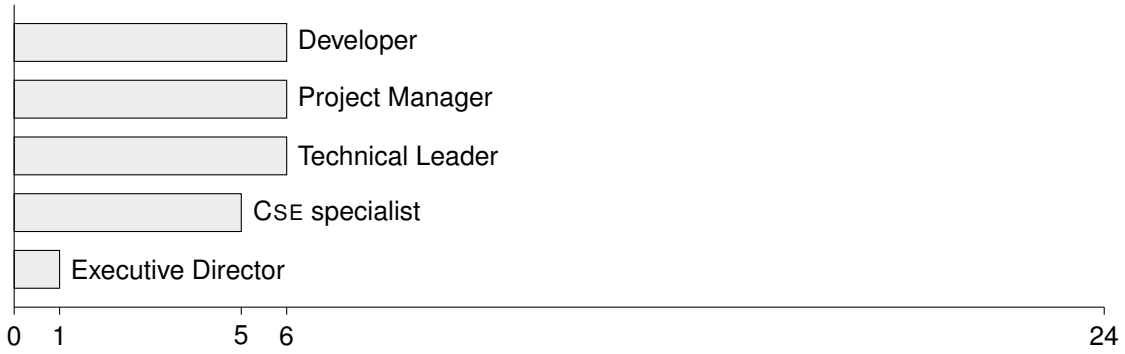


Figure 3.4: Number of practitioners distinguished by their role description.

We asked the practitioners whether they see themselves in one of the following three roles: *using*, *defining*, or *planning* CSE. Practitioners in a *using role* frequently apply and benefit from CSE, e. g., developers who regularly commit code. Practitioners in a *defining role* set rules on how CSE elements are applied, e. g., whether a code integration is triggered by an event, such as a commit, or on an hourly basis. It is the responsibility of a *planning role* to think ahead and take future addition to a CSE environment into consideration. Many practitioners saw themselves in multiple roles: 14 using, 15 defining, and 14 planning. Seven practitioners reported to adhere to all three roles. Six other practitioners grouped themselves into two roles at the same time: In one role, they collect knowledge regarding a CSE element and in the other role they share it with other practitioners. Six practitioners adhered to a single role only. Two developers saw themselves solely in a using role; a project manager and a CSE specialist classified themselves in a defining and planning role, respectively. Two other practitioners did not see themselves in one of the three roles provided. They and a third practitioner proposed an additional role: *promoting*. This role pushes CSE efforts forward, in particular in situations in which it does not appear reasonable from a cost perspective—but would pay off in the long run.

Projects

For each interview, we asked the practitioners to select one project that they are currently working on or which contains several CSE elements. It became apparent that the projects of several practitioners are similar throughout their department or division regarding the way they apply CSE. On average, 20.25 employees work in a project, for SMEs 10.00 and for corporations 22.81. Eleven practitioners (64.71 %, including all SMEs) consider their project as cross functional, e. g., involving several other stakeholders from within the company, such as marketing professionals to represent the users. Note that practitioners from all four SMEs stated a cross functional project structure. Three-quarters (15) of the projects develop *bespoke* software, e. g., custom software. Three projects (all by SMEs) develop commercial off-the-shelf software; two projects target both types. Figure 3.5 visualizes the practitioners' answers regarding the project's software type.

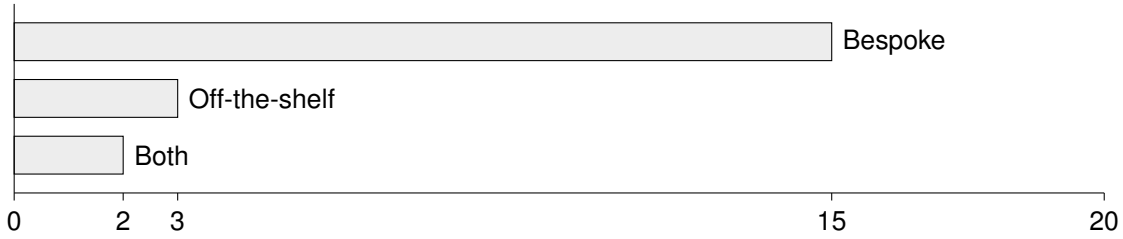


Figure 3.5: Number of projects distinguished by their software type.

We asked the practitioners, if their projects and the way they have to be approached need to comply with any legal, security, medical, or environmental factors; every second practitioner confirmed this. However, some practitioners referred to obligations that relate to the product, rather than the project itself.

3.5 Validity of Case Study Results

We conducted the interviews as part of a case study from a positivist philosophical stance [89], i. e., we tried to draw general conclusions on how practitioners apply CSE, as well as how they involve users and how knowledge would support their CSE process. As any study design, our interview study comprises several inherent limitations [298]; most notability, as we performed a case study, there is a lack of statistical generalizability, with no control over events that leads to a low precision of measurement [298]. In the following, we discuss the four criteria for validity for empirical research strategies as presented in Section 2.2.2, which apply for researchers with a positivist stance [89, 265]. In case the collection and assessment of data for a specific research questions posed an individual threat, we provide more details on it in the related section, such as for example in the case of Section 4.1.3.

Construct Validity

First, the practitioners resemble a heterogeneous group of individuals in which all of them have their own point of view on CSE. Chances are that the conformance between them are small. Also, the practitioners' roles and the project settings are context factors that influence their answers. We tried to address heterogeneity by describing observations and recommendations instead of facts that would require absolut measurements. Second, some of the questions may be interpreted by practitioners in a different way than intended by us. We tried to minimize this possibility by conducting two interviews with colleagues before we approached the practitioners. We discussed these interviews afterwards in a group of researchers to reveal potential misinterpretations. Also, the format of the interviews allowed the practitioners to ask questions at any time, which further reduces the likelihood of misinterpretations. Third, the researchers might have influenced the participants by asking questions in a specific way. To mitigate this risk, we used open-ended questions to elicit as much information as possible from practitioners. Finally, the collection of CSE elements introduced in Section 3.1 is based on a model that is an abstraction of reality and—to some extent—subjective. We asked practitioners to describe their experiences with the proposed set of CSE elements, which might have biased them. We tried to mitigate this risk by collecting additional CSE elements.

External Validity

First, we contacted companies that we already knew, such as through joint collaborations. This might have result in a sampling bias, i. e., the practitioners that we interviewed might not be representative of the target population. However, there is no central register of companies that apply CSE [302]. Clearly, this amounts to a risk to the representativeness of the participants. It is mitigated by the fact that the researchers are from two different universities, which increases the number of practitioners for contact. Further, the diversity of projects and participants that participated in the study reinforces the generalizability. Second, interviews are subjective, as they rely on the practitioners' statements, which restricts their external validity. To reduce subjectivity, we conducted 20 interviews, to acquire a broad spectrum of opinions. Third, the number of answers varied for most of the interview questions; if required for the interpretation of the data, we indicate this, mostly in the figures' captions. In some interviews, we skipped interview questions, either due to practitioners' time constraints or because they provided answers that made it clear that they cannot answer a subsequent question or an answer that rules out the meaningfulness of another question. However, we think that the diversity of projects and participants supports the generalizability of the answers.

Internal Validity

First, the practitioners might have provided answers that do not fully reflect practices of their daily work, as they were aware that the results would be published. We addressed this possibility by guaranteeing the full anonymity of interviewees and companies. Second, the interviewees' descriptions might deviate from what they actually do as it can be difficult for them to elaborate on their work practices in a retrospective fashion [89]. We mediated this aspect by stimulating verbose answers by providing examples to the questions we were asking. Third, the interpretation of answers might be biased by the researchers' *a priori* expectations and subconscious impressions. We addressed this threat by coding the transcriptions and discussing the codes. Finally, the slides that we shared with the practitioners before the interviews might have biased the practitioners' perception of CSE. However, we perceive this as a minor threat, since it only affects RQ 1; in addition, the practitioners might have prepared beforehand anyway.

Reliability

After we carried out the coding training and checked intercoder reliability, two researchers individually coded different transcripts, which might have affected the results. We mediated this threat by discussing questions during coding, especially in case of ambiguity. In addition, an additional researcher supervised the interview analysis procedure.

Chapter 4

Current Practice in Continuous Software Engineering

> *"Phenomenologists often work with interview transcripts, but they are careful, often dubious, about condensing this material. They do not, for example, use coding, but assume that through continued readings of the source material and through vigilance over one's presuppositions, one can reach the "Lebenswelt" of the information, capturing the "essence" of an account—what is constant in a person's life across its manifold variations."*
>
> — MATTHES B. MILES & ALAN M. HUBERMAN [209]

Continuous software engineering (CSE) as introduced in Section 3.1 bundles activities to enable continuous learning and improvement by frequently iterating on software increments [41, 109]. This classifies CSE as a software engineering process [146]. Krusche and Bruegge addressed the need for a formal description of the continuous aspects of CSE to enable its adoption in real world settings [179]. Similarly, researchers highlight challenges in the introduction and enhancement of CSE in companies [230, 297]. Practitioners need a starting point in order to approach CSE, since they might lack insight into interrelationships, potential risks, and challenges [252, 258]. Comparing their CSE process with what other companies have implemented allows practitioners to assess their own progress. Likewise, practitioners can benefit from guiding principles to establish CSE in their company [109].

We relied on the case study introduced in Section 3.3 as a means to collect insights to derive such guidance. We present results on the research questions introduced in Section 3.3.1.1 that address Knowledge Question 1. In the following, we report on practitioners' definitions of CSE (Section 4.1.1), CSE elements that they perceived as most relevant (Section 4.1.2), their experiences with CSE (Section 4.1.3), and future plans (Section 4.1.4). We discuss the results and create a model in Section 4.2 as well as present related work in Section 4.3.

4.1 Results on How Companies Apply CSE

We illustrate the results of the quantitative analysis of CSE elements and categories that were mentioned in the interviews in Figures 4.1 and 4.2. We address the RQs based on the data provided in both figures: First, we summarize the answer to a RQ. Then, we provide a more detailed analysis by stating observations.

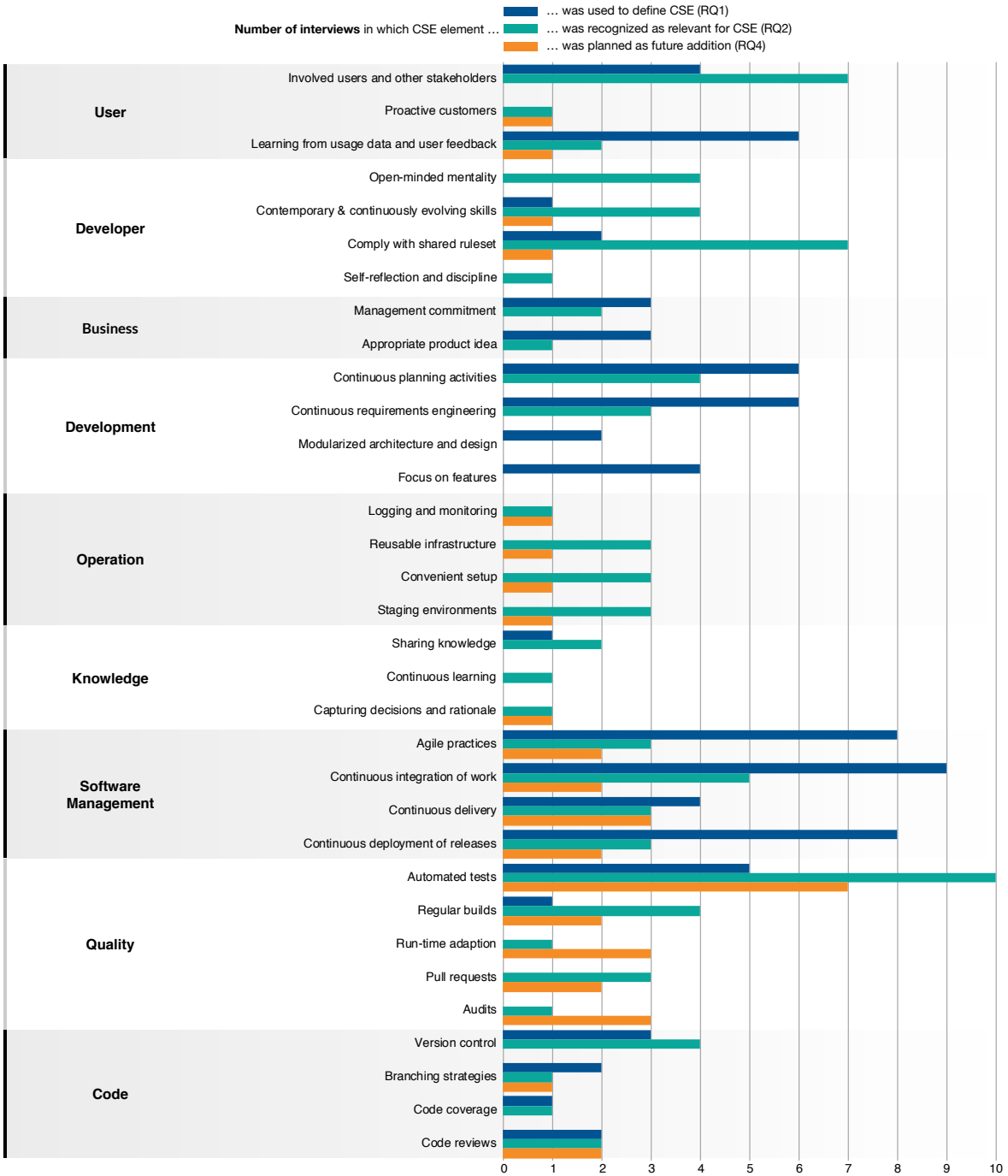


Figure 4.1: Number of interviews in which practitioners mentioned CSE element, separated by CSE categories and RQs. Adapted from Johanssen *et al.* [157, 158].

In Figure 4.1, the answers were summarized as one interview in case multiple practitioners participated at the same time. The CSE elements are headed by their categories. Blue bars indicate the number of interviews in which the respective CSE element was used to define CSE (RQ 1). Green bars indicate the practitioners' tendency toward relevant elements for CSE (RQ 2). Orange bars indicate CSE elements which practitioners noted as candidates for future additions (RQ 4).

4.1.1 Definitions of CSE

Since the term CSE only emerged in recent years, we asked the practitioners how they define CSE. We plot the results in Figure 4.1 using blue color.

> **Summary RQ 1:** We found that the practitioners' definitions of CSE are mainly driven by CSE elements from the software management category, i. e., continuous integration of work, agile practices, and continuous deployment of releases. CSE elements from the development and user category were also mentioned repeatedly. Based on the responses, we identified five perspectives that influence the definition of CSE: a tool-, methodology-, developer-, life cycle-, and product management perspective.

Out of the 24 practitioners interviewed, slightly more than half (54 %) were using the term CSE as part of their active vocabulary. About two-thirds (66 %) of all interviewees gave a definition of their understanding of CSE. Notably, 75 % of the interviewees in SMEs both gave a definition and actively used the term CSE. For some practitioners, CSE is still ambiguous: They describe it as *fuzzy*, *abstract*, and *lacking a distinction*.

Tool Perspective

Practitioners, i. e., six developers and CSE specialists, made use of tool descriptions when defining CSE. Their descriptions were characterized by statements such as *"in this regard, company A provides tool B"*, or *"after introducing tool C, we were able to accomplish element D"*, or *"we are currently looking into tool E of company F"*. Four practitioners explicitly highlighted that it is a well-chosen *tool chain* that enables CSE. In their opinion, the successful accomplishment of the steps availability, integration, and usage of tools allow a company to claim to be implementing CSE.

> **Observation 1:** In particular developers and CSE specialists rely on a tool-driven approach to define CSE. Commercial tools influence their understanding.

Methodology Perspective

In more than half of the interviews, practitioners cited a methodological perspective to define CSE. They emphasized a focus on short iterations and feedback. Not mentioning specific tools, many practitioners highlighted the importance of *how* the tools are applied. For instance, a sophisticated branching strategy should be preferred, instead of the exhaustive use of capabilities a version control system might offer. A state of *on-going iteration* should be reached, in which each commit leads to a finalized product. Different elements, such as continuous integration or agile practices, are applied to achieve a high level of automatization. Notably, some practitioners reflected on the combination of multiple CSE elements to achieve synergy effects. This implies that their perspective takes the impacts of CSE on other areas, such as the management of requirements, into account.

> **Observation 2:** Practitioners define CSE from a methodological perspective that aims for short iterations during software evolution. This perspective relies on well-defined steps in tool usage, workflows, and procedures. Every step enables a seamless workflow from a commit until its finalization in form of a build.

Several practitioners mentioned the importance of instant feature visibility to users. They reported that this enables constant retrieval of user feedback on the latest releases and more iterations with input from outside are performed. This is perceived as a major criterion for defining CSE by multiple practitioners. One practitioner advocated that every CSE process should be designed in accordance with the goal of matching customers' requirements. This can be achieved by keeping feedback loops as short as possible. Another practitioner advised the collection of feedback *"as often as necessary, rather than as often as possible"*.

> **Observation 3:** CSE makes changes instantly visible to users. As a result, user feedback can be elicited and used to match the software to the requirements.

Observations 2 and 3 are related to and depend on each other to reach their full potential. However, we observed that not all practitioners implement both, leaving opportunities for improvement.

Developer Perspective

Several developers, project managers, and CSE specialists suggested a developer-driven perspective on CSE. Similar to Section 4.1.1.2, they did not base their descriptions on specific tools. First, they emphasized that CSE enables developers to fully focus on their main task, i. e., developing software, rather than deal with other processes, such as infrastructure management. They included an increase in the speed of the development process by removing idle times. Second, practitioners

noted that CSE allows them to better estimate and classify their daily tasks. According to them, CSE ensures that newly introduced changes do not break code—an aspect which is not only in the interest of the overall product, but also a factor in the mind of developers. Therefore, they stated that providing a safe environment to develop and test software is a major characteristic of CSE. Third, the practitioners reported that CSE supports the detachment of recurring tasks from an individual; in particular, by introducing defined processes, knowledge vaporization can be prevented. Remarkably, one practitioner highlighted the increased responsibility of a developer when providing their definition of CSE. This related to the fact that developers independently create and deploy releases which—to unlock the full potential of CSE—should take place without any clearance or dedicated release plan.

> **Observation 4:** CSE allows developers to focus on their tasks and creates a safe development environment. Specific tasks are removed from individuals.

Life Cycle Perspective

Various practitioners agreed that CSE opens up a new perspective on the software life cycle: development, deployment, and operational phases blend into each other. Practitioners reported shorter intervals between the development and the production phases. They further stated that CSE is characterized by the fact that a system's functionality is extended continuously and shaped by what they referred to as a *continuous application life cycle management*.

Overall, practitioners attested CSE to follow clear structures, as they are established for instance by continuous integration or continuous delivery. One practitioner took the stance that—in order to support a full CSE life cycle—only certain elements need to be present, as long as customer requirements are reflected in the underlying strategy for the product life cycle.

> **Observation 5:** Practitioners characterize CSE by the blending of different phases of software engineering, such as development, deployment, and operation. According to their perception, this makes long-living systems easier to maintain.

Product Management Perspective

Project managers, technical leaders, and executive directors formulated a definition of CSE from a product perspective. In their opinion, CSE is represented by constant funding, provided to continuously improve a product. This included the project managers' ability to continuously acquire new requirements as well as to create and re-prioritize product tasks. One technical leader admitted that not every product is guaranteed to follow this pattern. According to this practitioner, the application of

CSE cannot simply be defined for a project: it is the product that determines whether CSE can be applied or not. Most of the projects are designed to follow other software evolution practices, and not CSE in particular. A product's compatibility with CSE processes needs to be ensured before applying CSE. Further, the environment in which the user receives the product plays an important role, as pointed out by a practitioner from a large corporation. It is required to keep pace with the CSE practices as defined in Observation 2. For instance, if the deployment of a product requires certain manual steps, the product itself cannot be developed using CSE processes. One practitioner defined CSE as the integration of customer, business model, software, and hardware. They refined their answer by stating that if a company successfully combines these four aspects, it is implicitly implementing CSE practices such as continuous integration and continuous delivery.

> **Observation 6:** Practitioners' definition of CSE is influenced by the product under development. Product-related factors such as funding, functionality, business model, and its future target environment need to match the continuous development capability.

4.1.2 Relevant Elements of CSE

We asked practitioners' for CSE elements that *drive* CSE. Thus, they were required to list the three—in their opinion—most relevant CSE elements. In case a practitioner mentioned a CSE element that was not part of Table 3.1, we recorded it as a relevant element. We plot the results in Figure 4.1 using green color.

> **Summary RQ 2:** Practitioners perceive CSE elements from three categories as most relevant: *quality*, i. e., automated tests, *user*, i. e., involved users and other stakeholders, *developer*, i. e., compliance with a shared ruleset. Practitioners mention more CSE elements: in particular the developers consider elements from the *code* category, such as version control, as obligatory, pivotal, and indispensable to any further steps in CSE. This strengthens the first stair in the *Stairway to Heaven* model of Bosch *et al.* [41]. We summarize the results as follows: user commitment, team commitment, and automated loop.

User Commitment

In particular practitioners from SMEs that develop off-the-shelf software highlighted the contact with users as a CSE element. One technical leader pointed out a significant difference in the user audience: While there is a large number of users that are *passively* using software, i. e., users that do not state an interest in new additions that do not affect their typical workflows, it is the less represented *active* user who

helps to make the CSE feedback loop efficient and functional. One project manager continued this thought by stating that the interaction with the users is barely technically defined in CSE. They stated that they approach this issue by actively trying to involve users through the promotion of nightly and beta builds. Two technical leaders claimed that the success of a CSE project depends to a great extent on the degree of user involvement—*"if there is no involved user, you lose"*. Some practitioners remarked that *"users do not know what they want until they see it"*. Enabled by continuous delivery, one practitioner noted that users can frequently provide feedback and thereby steer the development process toward their needs.

> **Observation 7:** Practitioners perceive the users' commitment toward taking an active part in the development process as a relevant aspect of CSE.

Team Commitment

Many practitioners perceived the team and its commitment as highly relevant for CSE. According to one developer, it is important that team members are open-minded toward the development process and take an active role in its formation. They need to adhere to a shared ruleset to work successfully. Practitioners illustrated a need for the full support of managers and executives. They should provide their attention to the project and trust in the performance and skills of the team. Rather than tools, it is the methods and processes introduced by agile practices that bind team members together. They proposed that a *continuous process improvement* activity should be carried out by the team.

> **Observation 8:** Practitioners perceive an open-minded team mentality that complies with a shared set of rules as the basis of successful CSE teams. Management commitment is indispensable, while agile practices serve as a unifying factor.

Automated Loop

Half of the practitioners declared the automatization of process loops to be the core of CSE. According to these individuals, discrete phases should be replaced by short, compact loops. They used the term *continuous pipeline* to describe a well-defined, highly automated process, which can be further adapted to the characteristics of the product under development. The practitioners shared a vision of a non-linear process that can either be serialized or else run in parallel. The *Automated tests* CSE element was assessed the most relevant by ten practitioners. Others mentioned continuous integration and continuous deployment as the major building blocks of an efficient automated loop comprising different fulfillment levels. Operational aspects, such as *staging environments*, completed their idea of an automated loop.

> **Observation 9:** Practitioners perceive a high maturity level of automatization essential for CSE. This is enabled by well-defined steps that form a non-linear process model. Practitioners state automated tests as the most relevant.

4.1.3 Experiences with CSE

We asked practitioners about positive, neutral, and negative experiences with CSE elements. Figure 4.2 shows the results grouped by their respective categories.

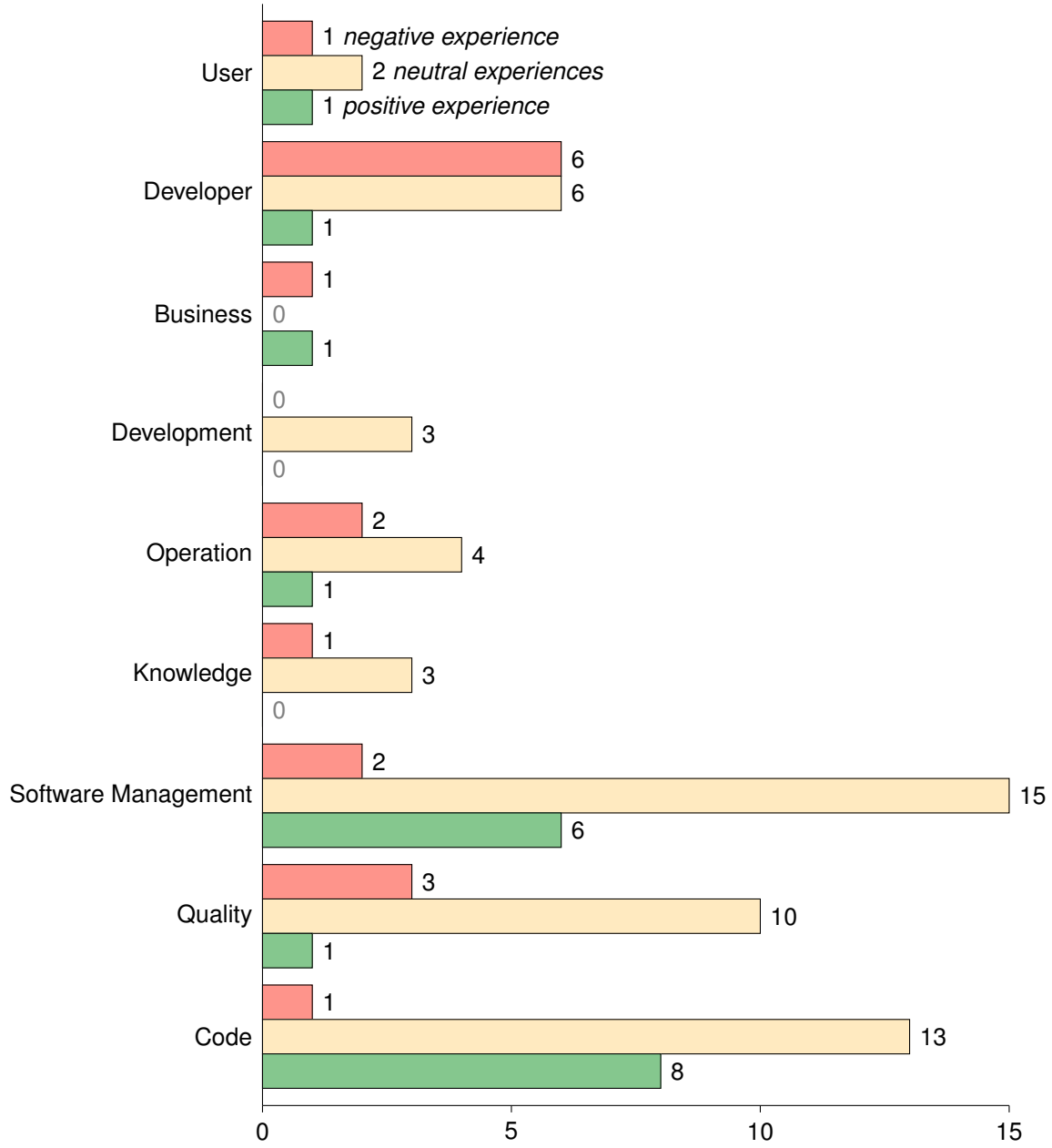


Figure 4.2: Number of negative, neutral, and positive experience reports by practitioners separated by CSE category. Adapted from Johanssen *et al.* [157, 158].

In Figure 4.2, the bar length indicates the number of CSE elements mentioned by practitioners. Note that not every practitioner provided an experience report and—given that we grouped the responses by the CSE categories—practitioners may be represented multiple times if they responded to more than one CSE element of the same category. Furthermore, we coded responses as either positive or negative only in cases in which clear indicators by the practitioners were provided, for instance if the practitioners used phrases such as *"we had problems with implementing <*CSE *element>"* or *"we could not work without <*CSE *element>"*. At the same time, the high number of neutral experiences could be understood as a questions-specific threat that is caused by the design of the question.

> **Summary RQ 3:** 19 positive, 56 neutral, and 17 negative experiences with CSE elements were reported. Notably, more than 50 % of the positive experiences were stated by practitioners from SMEs, while they formed about a quarter of the interviewee sample. Categories with many positive experiences as in *code* and *software management* are an indicator for CSE elements that can serve as an entry point to CSE, since they may be easy to implement. Few positive mentions as is the case with *knowledge*, *business*, and *user* may be a sign of the low maturity of CSE elements. Neutral responses may indicate that practitioners are currently evaluating various CSE elements in the field. A large number of negative experiences as with the *developer* category indicates challenging CSE elements. We discuss distinct experience reports derived from five CSE categories: developer, operation, software management, user, and quality.

Developer

Most negative experiences were reported in the developer category, in particular for *complying with shared ruleset*s and *contemporary and continuously evolving skills*. Negative experiences are amplified by problems with other CSE elements, such as *branching strategies*. One practitioner reported problems when dealing with too many branches, which they considered *"poisonous to continuous integration"*. However, they admitted that it is sometimes inevitable to have several branches, even though this situation can be approached with well-defined rules; for example, keeping the lifespan of a branch as short as possible, and committing code frequently. According to the responses of six practitioners, this demands attention from developers. Switching to a new way of developing software requires the willingness to evolve skills and extensive knowledge—which is why one practitioner viewed young graduates as having advantages over long-serving employees. One practitioner sketched solutions on how to overcome obstacles: providing incentives for successful work, enabling and supporting in-house training, as well as creating showcase projects. The practitioners agreed that an open-minded mentality on the part of develop-

ers, as well as their ability to adapt and withstand the speed and frequency of CSE amount to both the basic requirement and also a major challenge for developers.

> **Observation 10:** Practitioners acknowledge a major challenge in the developers' capability to comply with shared rulesets and in their open-minded mentality to continuously evolve their skills.

Two practitioners noted that automatization makes developers tend to use methods which they would otherwise dispense with. However, they admitted that this makes it also easier for developers to neglect other responsibilities; therefore, they stressed that CSE demands *"self-reflection and discipline"*. Practitioners with a leading role stated that they trust their team members. They initiate discussions to find a consensus whenever they think it becomes necessary.

> **Observation 11:** CSE does not solely build on the developers' skills, but also on their ability to reflect on their work and on their sense of responsibility.

Operation

One practitioner reported that corporations struggle with legacy burdens, e. g., existing tool contracts that are not intended for CSE. Similarly, tools intended for CSE are used for other purposes, such as issue tracking systems for internal incident management, rather than for software development. Practitioners of other corporations emphasized the fact that tools are not a hurdle for them, since they can easily be bought—it is the integration that poses the challenges. Other corporations have to adhere to formal regulations that impede or prevent CSE from being applied in a given project, e. g., by relying on paperwork-driven processes. They stated that they apply a *"pseudo-scrum"* approach. One practitioner stated that there is a risk that agile projects could fall back into their previous static patterns.

> **Observation 12:** While practitioners are willing to apply CSE, it is their company's current set of tools that keeps them from making a complete transition and fully adapting CSE. Furthermore, requirements in regulated domains hinder the implementation of CSE.

One practitioner complained that a major cost factor lies in setting up the infrastructure for new projects to use CSE elements. Furthermore, given the internal hierarchical and management structure, they report that their corporations do not have the capacity to respond rapidly to changes within the project.

> **Observation 13:** Practitioners state that the successful implementation of CSE requires the ability to set up projects without major cost or time penalties.

Software Management

Practitioners reported positive experiences with the implementation of *agile practices*. Building-blocks such as sprints, review meetings, or SCRUM-Boards are well-received and provide high value. Some difficulties arise during task prioritization, since only limited resources—time and money—are available. Apart from that, CSE elements, such as continuous integration, are essential to practitioners. One practitioner mentioned significant synergy effects when using tools of the same vendor for issue tracking, source code management, as well as continuous integration and delivery.

> **Observation 14:** Practitioners attest that CSE elements related to software management, such as agile practices or continuous integration of work, are widely and successfully adopted in their projects.

User

CSE elements related to users were barely referenced whenever practitioners are asked about their experiences. The user's role was rated as *fuzzy* and there is a threat that user feedback does not continuously flow back to developers. One practitioner ascribed this to the fact that CSE does not produce major releases that are perceived as notable changes by the users. Thereby, user feedback is submitted late in the process in the form of incidents or change requests. Developers then lack traceability links to changes that caused the feedback. One project manager was concerned that software quality suffers from the release frequency in the short run since immature releases impede the users' confidence in the product.

> **Observation 15:** Practitioners have not yet created processes that interact with the users in a way similar to well-established practices such as continuous integration. This is mainly due to the fact that the users' responses to ongoing changes are difficult to record, trace, and assess.

Quality

Practitioners welcomed CSE elements such as *pull requests* combined with *code reviews*. *Code coverage* and *audits* are reported as practices that are gaining in importance. However, some practitioners raised concerns because quality metrics are not

being tracked. We observed that practitioners' responses regarding the *quality* category are driven by various testing and exploration reports. First, every project strives for high software quality and therefore tries to invest effort into improving. Second, as there is no final release, processes to improve software quality can always be developed further. Third, the influence of changes to software quality might become apparent only at a later time.

> **Observation 16:** Practitioners have had varying experiences with quality elements during CSE, but they still invest into improvements.

4.1.4 Future Plans for CSE

We asked practitioners which CSE elements they plan to add in the future to discover future trends in CSE. We plot the results in Figure 4.1 in orange.

> **Summary RQ 4:** Practitioners' plans are vague and mostly distributed across elements. 19 CSE elements either received only one, two, or three mentions by the practitioners in the interviews. One CSE element stood out with seven mentions: *automated tests*. We found that the majority of practitioners described plans that span multiple CSE categories. We identified three main strategies in the practitioners' answers: enhancement, expansion, and on-demand adaption.

Enhancement Strategy

Practitioners based their strategy for the future on a combination of the *methodology perspective* as described in Observation 2 and *quality*, one of the most relevant categories mentioned (Observation 9), yet one with mostly neutral experiences (Observation 16). Seven practitioners mentioned automatization in the context of quality as one of their major plans for the short and long term. While *automated tests* are applied for some parts of the products, they should be made available for all. Two practitioners described their plans of combining elements from the *operation* category, such as deployment in containers to enhance automatization. Three practitioners listed activities to enhance their current state: code quality workshops, giving code metrics a meaning by calling for action rather than representing read-only information, and connecting CSE elements from different CSE categories.

One technical leader elaborated on their planes to bring the interaction with the user to the next level by detaching feedback collection from the individual—which is currently often the case—and creating a well-defined, high maturity level process similar to the one used for continuous integration. Other practitioners mentioned various ways of optimizing the implementation of agile practices or the application of branching strategies.

> **Observation 17:** Practitioners aim for a fully automated loop to increase software quality by applying goal-driven enhancements to existing CSE elements.

Expansion Strategy

The future plans of four practitioners can be summarized as an *expansion strategy*, i. e., applying recently established CSE elements to other areas of a project. The expansion of continuous delivery to more platforms was mentioned several times. This means adapting similar practices such as automatic deployment in mobile environments to their server-side counterparts. Similarly, expanding continuous integration to more platforms was mentioned several times; for example, one practitioner praises progress in *Java* environments, while they struggle with *JavaScript*. Another practitioner stated the expansion of documentation to more areas than it is the case at the present time.

> **Observation 18:** Practitioners aim to extend efficient CSE elements to other areas of the project or similar products.

On-Demand Adaption Strategy

Three practitioners indicated a general interest in future CSE additions, however, they rely on an event-triggered or *on-demand* strategy to adapt, i. e., enhance or extend, their CSE elements. One practitioner described an exploratory process in which CSE elements are added *step by step*: If they encounter a situation that would benefit from improvements, they initiate further investigations into possible solutions. In general, they stated that main criterion for automatization a process needs to be performed several times manually before they consider it for improvement. Another practitioner made the addition of further CSE elements dependent on the team's dynamic. They elaborated that the addition of more members to a team introduces a more hierarchical order of roles that might slow down CSE development, while a small-sized team tends to be more explorative and more eager to try out new things. A project manager highlighted the effort in time that is required to implement certain CSE elements, which makes an overall process transition a time-consuming undertaking. One practitioner mentioned that in particular the lack of know-how hinders leads to an adoption only when it is necessary.

> **Observation 19:** Practitioners make enhancements and additions to CSE dependent on events that call for action. They postpone decisions for further additions to a later point in time.

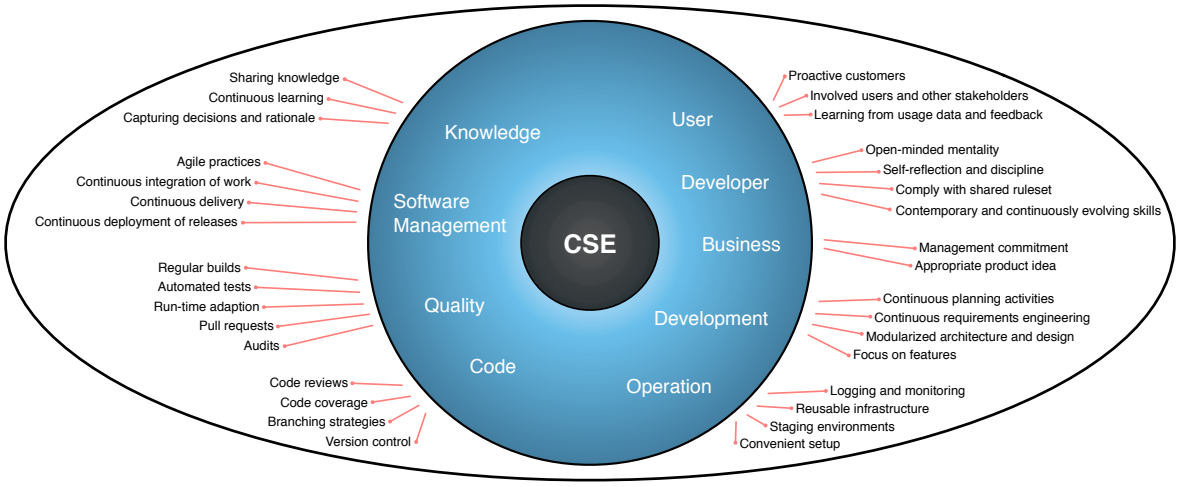


Figure 4.3: Visualization of the *Eye of CSE*, adopted from Johanssen *et al.* [157, 158].

4.2 Discussion of Results

In this section, we summarize our insights in the form of a model and present related work. Above all, we find it necessary to recapitulate that some observations from this study appear obvious, ambiguous, or may even be seen as a tautology. However, since we are searching for empirical evidence on which to base assumptions that were previously only anecdotal, to some extent this leaves us with reporting *"obvious conclusions"* [302].

Based on the practitioners' answers, we updated the CSE elements and categories in Table 3.1 by adding three new categories—*developer*, *business*, and *operation*—and removing two CSE elements, which are represented in the new categories. We refer to the final set of CSE elements and categories as the *Eye of CSE*. A well-established infrastructure as demanded by our CURES vision (Section 3.2) incorporates all of these CSE elements. We provide a visual representation in Figure 4.3.

The model in Figure 4.3 consists of nine CSE categories and 33 CSE elements. The proximity of elements and categories suggests relationships between them. The model can open up one's eyes to new ideas for additions to current CSE processes.

The eye's focus lies on a comprehensive implementation of CSE, represented by the pupil. The process of reaching this goal is influenced by the categories that are part of the eye's iris. The categories define the diameter and size of the pupil and thereby the perception of CSE. We learned from the interviews—in particular from practitioners' future plans—that CSE categories are intertwined and have fuzzy boundaries. This is partly different from the sequential nature of the *Stairway to Heaven* by Bosch *et al.* [41, 230]: even if some CSE elements, such as continuous integration and delivery, require a stepwise introduction, the practitioners' statements suggest that CSE should be approached from multiple angles simultaneously. From the results of our interview study, we cannot advise a clearly defined sequence of

adding CSE elements toward the implementation of CSE in a company. Instead, the *Eye of CSE* can serve as a checklist for practitioners to tackle the subject of CSE by incrementally applying CSE elements and keeping an eye on potential next steps.

During the design of the *Eye of CSE*, we strived to accurately allocate CSE elements to categories by analyzing the practitioners' answers and carrying out internal discussions. By connecting CSE elements to the iris and not directly to CSE categories, we acknowledge that one CSE element can relate to one or more categories—their allocation is rather loose. Therefore, the proximity of CSE elements within the *sclera*, the white of the eye, only suggests a relation to a category and among multiple CSE elements. The grouping of CSE categories allows practitioners to recognize relationships based on their proximity within the eye and the position of the CSE elements. In particular, when improving one category, it can be worth considering the addition of another category that features similar CSE elements.

> **Example 7:** Whenever practitioners expand on *software management*, the categories *knowledge* and *quality* should be incorporated, since their containing CSE elements are interrelated and have a positive effect on each other.

The addition of one CSE element from a category may have a positive effect on multiple CSE elements from the same or other categories. The combined addition of CSE elements allows companies to benefit from synergies of the CSE elements' effect or to reduce the implementation efforts. We tried to arrange related CSE elements and categories closely in spatial proximity. However, this relationship cannot be derived from the model at its current state. Therefore, we suggest that practitioners concurrently improve CSE elements from one category as well as across category borders to benefit from their synergies or to reduce the implementation efforts.

> **Example 8:** *Audits* can include *code reviews*, which makes the element being part of the *quality* category. Likewise, *learning from usage data and feedback* requires the developers' *open-minded mentality*. Addressing *sharing knowledge* is an example for a CSE element that has positive effects on almost all other CSE elements, for instance *agile practices* are improved through experience reports or *branching strategies* from best practices.

The model should open practitioners' eyes to new ideas when extending their CSE process. Furthermore, the relationships can start discussions for the future consolidation of the model: The categories *user*, *developer*, and *business* could be further summarized as *stakeholders*, while the categories *business*, *development*, and *operation* share common characteristics and might be combined as *BizDevOps*, following the naming conventions by Fitzgerald and Stol [109]. As noted, we see structural dependencies between *software management* and *knowledge* and between *quality* and *code*.

4.3 Related Studies

To the best of our knowledge, there are no previous studies with practitioners that address CSE as a process. Therefore, in this section, we present and discuss studies that followed research approaches similar to our work and addressed specific CSE elements and CSE categories.

Kuhrmann *et al.* researched development approaches in practice [181]. They highlighted the importance of traditional frameworks, a factor that is supported by our Observation 12. Their results indicate that companies apply hybrid approaches, which are defined stepwise, in ways similar to the strategies we identified in Section 4.1.4. Gregory *et al.* investigated challenges that agile practitioners are facing with methods that included interviews [126]. Their results are similar to our observations. They identified seven interrelated themes of challenges regarding claims and limitations, organization, sustainability, culture, teams, scale, and value. Likewise, we reported on practitioners' experiences in CSE including challenges. Mäkinen *et al.* reported the widespread adoption of version control and continuous integration using semi-structured interviews with 18 organizations [198]. Our observations confirm this situation in practice. In fact, we found that most positive experience reports were stated in the respective CSE categories, as it is shown in Figure 4.2. Ståhl and Bosch interviewed practitioners to assess their experience of continuous integration and discuss benefits [295]. We can confirm most of their findings, e. g., that it increases developer productivity, as described in Observation 4. The same observation partially supports their finding that practitioners see improvements in project predictability. Further literature studies list benefits and challenges regarding continuous integration, delivery, and deployment [258, 286]. Kevic *et al.* revealed the positive impact of experiments; this points toward a relationship between continuous deployment and a change in user behavior [165], and supports our model presented in Figure 4.3. Dybå and Dingsøyr highlighted human and social factors in work settings that are similar to CSE [87]. Ayed *et al.* concluded that the success of agile practices depends on various social and inter-cultural factors [18]. Larusdottir *et al.* noted the importance of teams' ability to trust in their capabilities [184]. For our initial election of CSE elements in Section 3.1, we did not take similar aspects into consideration. However, based on these reports and multiple answers in our study, we added the category *developer*, as described in Section 4.2.

Chapter 5

User Feedback in Continuous Software Engineering

> *"The computer 'user' isn't a real person of flesh and blood, with passions and brains. No, he is a mythical figure, and not a very pleasant one either. A kind of mongrel with money but without taste, an ugly caricature that is very uninspiring to work for. He is [...] such an uninspiring idiot that his stupidity alone is a sufficient explanation for the ugliness of most computer systems."*
>
> — EDSGER W. DIJKSTRA [81, 82]

Continuous software engineering (CSE) has enabled new capabilities to capture insights, for instance regarding decisions [167] or user involvement: Continuous delivery promotes user feedback on the latest changes to a software increment [176, 178]. While user feedback can be collected in an explicit form, such as written reviews, the monitoring of implicit user feedback improves understanding the need for new requirements [196]. CSE forms the basis for continuous experimentation, which opens new possibilities for requirements engineering [273]. Reports on current practices of how user feedback is captured and utilized in industry are sparse. While many practices, such as continuous integration [208, 296], continuous delivery [60, 186], or their combination [87, 286] have been extensively researched, the interaction with users in CSE environments is less understood. Rodríguez *et al.* identified a research gap for mechanisms that make use of feedback [258].

We are interested in understanding current practices and identify possible improvements. Using the case study described in Section 3.3, we strived to address the research questions introduced in Section 3.3.1.2 that address Knowledge Question 2: in Section 5.1, we present results for all sub-RQs. Based on the practitioners' answers, we derived five recommendations that can guide improvement of continuous user feedback capture and utilization and describe them in Section 5.2, followed by related studies in Section 5.3.

5.1 Results on User Feedback in CSE

For each sub-RQ, we present the practitioners' answers as a bar chart and detail them in an explanation. All charts extend to 20 interviews to enable direct visual comparisons among them. If an answer for a sub-RQ received two or more codings, we visualize the count as a grey bar and do not reduce it from the individual coding to maintain their expressiveness. In such cases, we add the number of answers that were exclusively (*excl.*) applied to a coding inside their respective bar. We also provide a summary at the beginning of each results section that provides an overview of the respective RQs and note observations that stood out.

5.1.1 User Involvement in CSE

With RQ 5, we intended to investigate which user feedback types practitioners rely on for the capture and utilization and to which artifacts they relate the user feedback.

> **Summary RQ 5:** Practitioners predominantly rely on explicit user feedback; implicit feedback is never the single source of feedback. With respect to the relationship that practitioners use for user feedback, no artifact is standing out.

Types of User Feedback

In Figure 5.1, we visualize the practitioners' answers regarding sub-RQ 5.1.

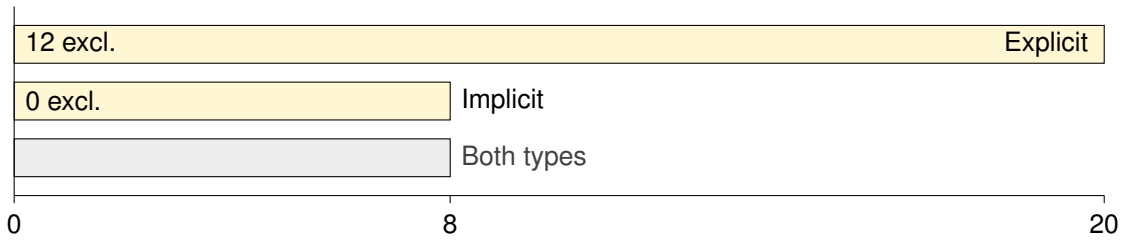


Figure 5.1: Number of interviews in which practitioners addressed user feedback types. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

All 20 practitioners reported that they rely on *explicit* user feedback. Four practitioners did not further specify the kind of feedback, besides that it originates from direct contact with the user. Five practitioners detailed how the explicit user feedback is produced, which allows conclusions about its kind. They described personal meetings in the form of presentations or design thinking workshops, in which they showed the user the latest version of the application. They collected either written or verbal feedback, using questionnaires or manually noting down how the users interact with the application. Three practitioners relied on explicit user feedback that is provided in the form of issue tickets. In some cases, the issue management

system is publicly available. Then, the users themselves can create feature requests and bug reports that are directly passed to the developers. In other cases, the issues are reported by an intermediate layer such as first-level support.

None of the practitioners stated that they rely solely on *implicit* feedback; however, eight practitioners described the use of both types, partly depending on when and from whom they capture the user feedback. The practitioners noted to start with the capture of explicit user feedback at an early stage. This may include video recordings, surveys, user interaction with prototypes in a controlled environment, i. e., applying observation techniques, as well as email and textual feedback that can be enriched with screenshots to receive qualitative, i. e., explicit, user feedback. After reaching a certain maturity level, they transition to collecting implicit user feedback and apply quantitative methods: the practitioners have mentioned A/B tests and the collection of usage statistics and analytics data.

One practitioner mentioned that they enrich the frequent implicit feedback by acquiring explicit user feedback through channels such as service hotlines or events such as exhibitions. Another practitioner mentioned primarily focusing on implicit user feedback, which allows them to gather information on how and when a user interacted with the application. They then enrich the data with explicit feedback.

> **Observation 20:** Practitioners rely on explicit over implicit user feedback.

Artifacts for User Feedback Relationship

In Figure 5.2, we visualize the practitioners' answers regarding sub-RQ 5.2. In six interviews, no answers were provided.

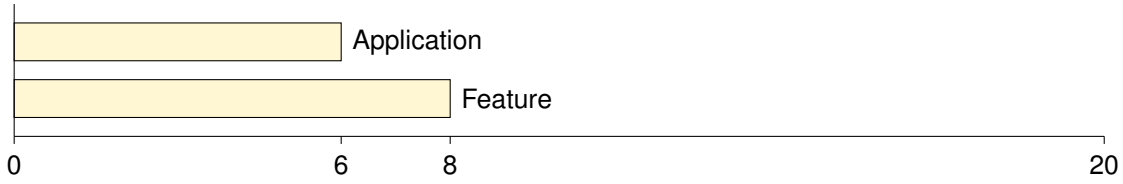


Figure 5.2: Number of interviews in which practitioners addressed artifacts to which they relate user feedback. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

Six practitioners stated that they relate user feedback to the *application* itself, which serves as a high-level reference point. Two practitioners relate user feedback to operational or technical aspects of the application; they are only interested in the outcomes, i. e., what the application was used for. Two other practitioners stated that individual features are not of interest to them, that they instead are interested in the overall acceptance of the application. For individual feature aspects, they stated that there are always formal specifications that they fulfill and guarantee, such as by using test cases. Therefore, they do not require additional verification through user

feedback. This perspective is continued by one practitioner who emphasized that they are interested in finding situations in which users behave differently from what was expected—which could occur at any time while using the application. Another practitioner reported to follow a similar approach, which is, however, driven by the question of how the overall application could be improved.

Eight practitioners stated that they collect user feedback with a focus on the *feature* that is currently under development. Following an example by the practitioners, that may be a user feedback on how a new menu navigation is adapted by users. Five of them detailed how they establish the relationship between user feedback and the feature under development: One practitioner pointed out that this can be achieved by explicitly asking for feedback in a survey that allows users to describe the feature, which requires them to verbally relate their feedback. Another approach is having the user add reference points that guide developers to the feature in question; for example, by attaching screenshots to the provided feedback. The practitioners explained that without such guidance, reproduction of the feedback and its understanding are difficult—a situation they find often when feedback is received from external users. To relate implicit user feedback to a feature, one practitioner reported that they create dedicated profiles for individual users to map a change in the feedback to the increment. One practitioner noted that they consider feature-specific user feedback, but only in an isolated surrounding which is meant to qualitatively improve a particular feature, rather than comparing it with an alternative implementation. Another practitioner explained that their features correlate with services their company provides. Based on this relationship, they conclude that—if users activated a service—they also made use of a particular feature. This enables a high-level assessment if and how many users interact with a feature.

> **Observation 21:** Practitioners struggle to relate user feedback to a feature.

5.1.2 Capture of User Feedback

With RQ 6, we intended to investigate the frequency of user feedback capture, practitioners' use of tool support, feedback sources, and whether they capture the context of user feedback.

> **Summary RQ 6:** Practitioner primarily capture feedback following well-defined events or periodically, rather than implementing a continuous approach. Most of them have tool support in place, however, processes that rely on manual capture are wide-spread. User feedback from external sources prevails internal sources, however, latter ones are perceived as highly beneficial. Practices to capture context of user feedback are rare and immature.

Frequency of User Feedback Capture

In Figure 5.3, we visualize the practitioners' answers regarding sub-RQ 6.1. In three interviews, no answers were provided.

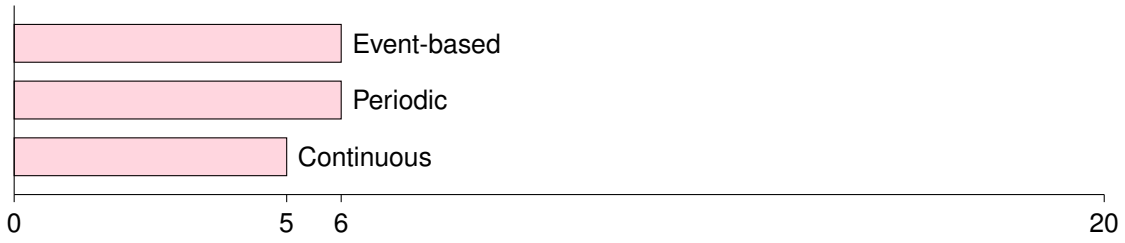


Figure 5.3: Number of interviews in which practitioners addressed how often they capture user feedback. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

Six practitioners reported to capture user feedback *event-based*, which is reflected in a sporadic or unscheduled process. Three practitioners stated to capture the feedback ad-hoc or on-demand. This applies when they initiate the capture process on their own in case they require more insights. Likewise, two other practitioners mentioned making the capture process dependent on when the user provides explicit feedback after using the application. One practitioner described events that relate to a well-defined point in time, such as the beginning or end of the application development. This could also include the capture of user feedback that is triggered by external changes, such as updates of the operating system of the device running the application. Another practitioner highlighted that the capture of user feedback comes down to individual, rare, and irregular personal meetings.

Six practitioners collect user feedback in a *periodic* manner, i. e., repetitively or iteratively. They mentioned periods with a minimum length of two days; to one, two, and three weeks; up to a month. One practitioner criticized the testers always lagging behind the latest development, which leads to a time delay between feature development and the capture of user feedback. For this reason, the practitioner referred to the capture process as semi-agile. Another practitioner reported pilot phases that are dedicated to the capture and understanding of user feedback.

Five practitioners stated that they *continuously* capture user feedback. Only one of them continuously collects explicit user feedback, in the form of frequent meetings, phone calls, and video conferences with the users, which almost always lead to immediate responses. All of the other practitioners rely on implicit user feedback during their continuous capture process. Notably, the five practitioners spoke with one voice in concluding that—while the user feedback is continuously captured—this does not imply that it is assessed or utilized; nor is it used to augment decisions or derive new insights and requirements.

> **Observation 22:** Practitioners predominantly capture explicit user feedback following event-based and periodic approaches. The majority of continuous approaches focus on implicit user feedback.

Tool Support for User Feedback Capture

In Figure 5.4, we visualize the practitioners' answers regarding sub-RQ 6.2. In two interviews, no answers were provided. We defined tool support as a system designed to capture user feedback in the first place.

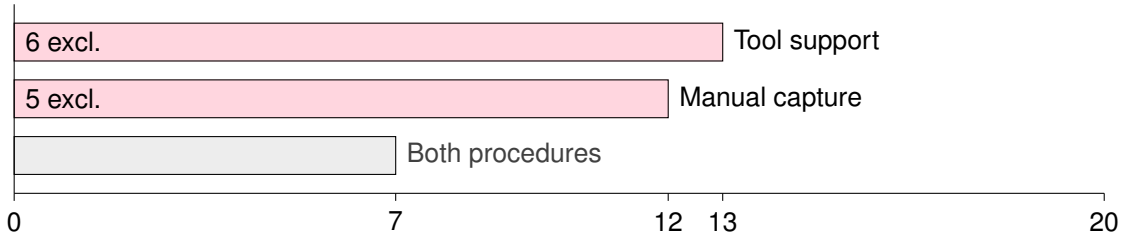


Figure 5.4: Number of interviews in which practitioners addressed tool support to capture user feedback. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

Overall, 13 practitioners rely on *tool support* to capture user feedback. All of them reported using standard tools for user feedback capture; they listed various tools that are known for their analytics and experience analysis capability, of which are prominent products from the following companies: Google (4 mentions), Adobe (3 mentions), and Microsoft (2 mentions). Four other practitioners listed tools such as Atlassian Jira or Redmine to capture explicit user feedback reports. On the one hand, it enables them to directly utilize the feedback for prioritization of requirements and development tasks. On the other hand, it helps them in finding major problems at times of peak activity. One practitioner stated that, while the possibility to use a particular tool exists, it is not being used. Besides standard software, 5 of the 13 practitioners additionally rely on custom software, i. e., tools that they developed for the sole purpose of capturing user feedback while their particular application is in use. Note that all of the five practitioners that use customized tools also employ customized tools to overcome limitations of the custom software: This enables them to track individual features, as well as other aspects, such as monitor the system's run-time behavior of the system. This may be the intensity of the workload that is managed by server-side components of the application. This helps practitioners make decisions on how to proceed with development. Practitioners also create custom scripts to track and understand users' adaption of their features.

We categorized 12 responses as *manual capture*, in which practitioners described procedures that require no software at all, or only tools that are not primarily designed for the capture of user feedback. This includes practitioners' relying on pen

and paper, on meetings or workshops that allow verbal exchange with respect to a prototype, or on indirect contact with end users via a product owner who serves as a proxy. Likewise, explicit user feedback capture via Skype or video recordings were mentioned by two practitioners. Two other practitioners stated to manually capture and externalize observations in wiki pages that are used for knowledge management, such as on Atlassian Confluence. Three practitioners highlighted that they rely on intensive email interactions to capture user feedback. Finally, two practitioners rely on online tools, such as Google Forms, that allow them to conduct surveys to capture user feedback and initiate interviews.

> **Observation 23:** The manual capture of explicit user feedback reduces the efficiency of an ongoing development process.

Sources for User Feedback

In Figure 5.5, we visualize the practitioners' answers regarding sub-RQ 6.3. In two interviews, no answers were provided. We distinguish between internal and external sources of feedback. The developers themselves or team members are internal sources, whereas external sources are stakeholders, i. e., customers or end users. We describe the answers in the respective explanations if they address both sources.

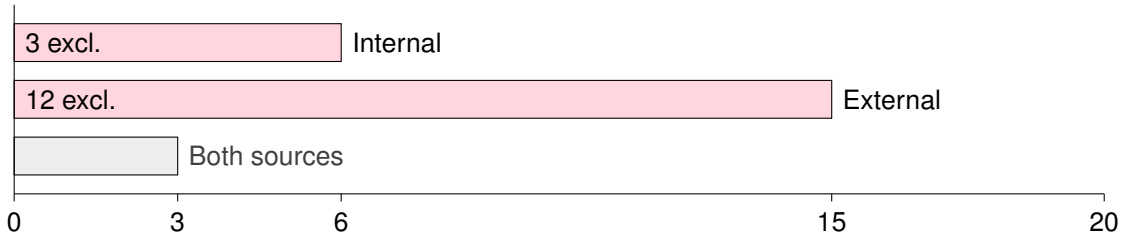


Figure 5.5: Number of interviews in which practitioners addressed sources for user feedback. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

Six practitioners mentioned user feedback capture from *internal* sources. One practitioner explicitly highlighted user feedback capture from other departments as a key factor for successful implementation. In their case, their marketing department always uses the latest version of the application under development and thereby becomes an internal customer. This also allows them to directly discuss the feedback in case something is unclear, either via instant messaging or direct "water cooler talk", mostly on a sporadic basis. Another practitioner emphasized the ability of individual developers to consider on aspects such as making the application easier to understand. Also, they pointed out the importance of having responses from users that only interact with their system sporadically. They summarized that a trusted relationship with fellow colleagues allows exchange of honest feedback.

Three practitioners stated that they rely on multiple extensions of the development team, which include dedicated teams that focus on testing, providing feedback, and suggesting improvements. One of those practitioners noted that the decoupling of requirements implementation and testing results in a waterfall-like, non-agile procedure, which, however, might also be related to their product context. The two others described user interface-focused teams that are generally able to provide instant feedback. However, dependent on the extent of the changes, they may not be able to keep pace with the development process and therefore lag behind. The developers consequently receive delayed user feedback, which tends to increase their workload. In the worst case, new feature improvements build upon features that have not yet been tested or validated by user feedback.

Fifteen practitioners addressed user feedback capture from *external* sources, i. e., a focus on external customer or actual end users. For three practitioners, this means sitting down with the customer and presenting the latest changes to the application under development and to collect user feedback. According to them, this requires tool support and typically results in high capture and assessment efforts. However, through the personal contact, the customer directly provides feedback to the developer in charge. The interpersonal aspect, i. e., an increased sense of responsibility by the developer and a high appreciation by the customer, was rated as important by the practitioners, as it contributes to the quality of feedback.

Four practitioners noted that the way they deal with user feedback depends on the reporting source. While they review user feedback from time to time, they continuously capture it from external sources without a particular goal in order to be ready to provide on-demand insights when requested. Three practitioners noted to capture user feedback from sources depending on their current development progress. For them this means to start with paper prototypes and collect explicit user feedback in the beginning and transition to more elaborate capturing of implicit user feedback as the product matures. Two other practitioners described a phase-driven capture process, whereby a limited number of users, or the customers themselves, are actively using the latest version of the application, in what is called a beta or pilot phase, while the practitioners are capturing user feedback. Two practitioners reported to focus their user feedback capture process on clearly defining a test scenario which they use to investigate a particular aspect of the feature under development. Three practitioners stated that they follow the provided specification and rely on user feedback that is handed to them by the customer or the product owner and thereby externalize the capture process to another stakeholder. Finally, three practitioners introduced a follow-up approach to capture user feedback; after they implement a change, they actively request feedback from sources they expect to be future users or who have requested a similar functionality before. They inform them in case something is changed on the basis of their feedback.

> **Observation 24:** Practitioners gain many insights by retrieving user feedback frequently from internal sources, such as fellow team members or colleagues from other teams.

Context of User Feedback

In Figure 5.6, we visualize the practitioners' answers regarding sub-RQ 6.4. In 12 interviews, no answers were provided.

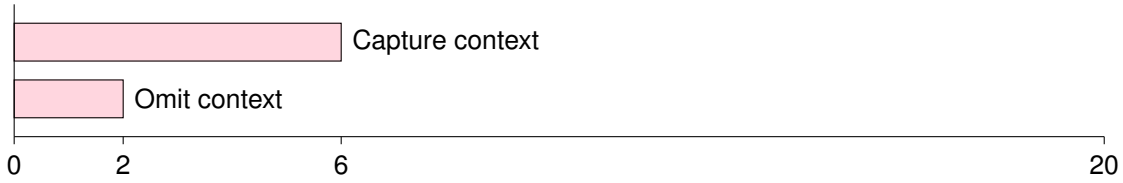


Figure 5.6: Number of interviews in which practitioners addressed context of user feedback. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

In total, six practitioners stated to *capture context* of user feedback. One technical leader described their option to inspect the environment in which the users interact within the application by collecting different kinds of sensor data. Another project leader, who pointed out a less technical, yet marketing-related view of the users' context, described that insights such as a users' purchase history allows them to get a deeper understanding of possible needs and the user environment. A developer noted that context information that allows them to derive helpful insights may be as simple as the local time, indicating whether the application is used during lunch break or typical working hours. Another practitioner reported that the time may also be used to systematically segment user feedback. This supports both a quantitative and qualitative user feedback comparison. Another practitioner stated that the captured context is similar to a snapshot, rather than a comprehensive user profile. Two practitioners acknowledged to capture, but do not use context information.

While two other practitioners stated that they consciously *omit context* capture, only one of them provided a reason: They primarily rely on user feedback from fellow team members and colleagues; formal regulations, company guidelines, as well as works councils and unions apply legal limits to what information may be captured. In particular, monitoring additional data to capture the context could reveal personal data, such as individual preferences.

> **Observation 25:** Context information provides valuable information to a few practitioners, however, its capture process requires multiple considerations beyond technical aspects.

5.1.3 Utilization of User Feedback

With RQ 7, we intended to investigate reasons why practitioners capture user feedback, their ability to exploit changes over time, and whether they combine different types of user feedback.

> **Summary RQ 7:** Practitioners' utilization of user feedback is distributed across the use for planning, support, and improvement activities. The practitioners rarely exploit the change of user feedback over time. Similarly, practices to combine user feedback from different sources do not follow systematic approaches.

Reasons for User Feedback Capture

In Figure 5.7, we visualize the practitioners' answers regarding sub-RQ 7.1. In eight interviews, no answers were provided.

Figure 5.7: Number of interviews in which practitioners addressed why they capture user feedback. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

In seven interviews, practitioners provided answers that are linked to *planning* activities during different phases of an application development cycle. Two practitioners noted to capture user feedback with the goal of performing business modeling, e. g., exploring new technologies, and defer other utilization of the user feedback, such as usability engineering, to a later point of the project. One of them added that, while there is a need to draw conclusions from user feedback particularly during early phases of product development, the overall goal should be diversification based on user feedback, which is provided for different reasons. As a result, user feedback should be considered but not necessarily cause fundamental changes to the project. Four practitioners stated that they utilize user feedback to prioritize features. For two of them, this primarily means deciding which feature needs to be implemented first or with a higher urgency, while two others use this insight to remove old features. Notably, one practitioner indicated the possibility to also detect the need for new features. One practitioner emphasized the possibility of assessing the acceptance of a feature from a marketing perspective. This allows conclusions

to be drawn regarding business-focused metrics, such as answering the question of whether invested efforts were worth it.

Six practitioners reported a use of user feedback with a main focus on *support* activities, such as bug fixing. One of them called this a customer-centered view. Four practitioners outlined that, when user feedback is used to identify bugs, it helps them to draw conclusions about the problem, e.g., by extracting and reconstructing the flow of the events that caused an error. One lead developer emphasized that implicit user feedback in particular is used to ensure that the application and its functionality are working as intended.

Five practitioners described user feedback utilization with the goal of *improving* the quality of a feature and as a decision-support for doing so. For another practitioner, this means to continuously monitor and observe the way the changes are adapted by users, which helps to develop a gut feeling of how the software is performing. Another practitioner observed a change in how they utilize user feedback; they are currently transitioning from a focus on reconstructing errors toward developing an understanding of how the users use the application. This will help them to assess whether the users are using the feature as intended. Two other practitioners continued this idea and stated that sometimes the wrong requirements are used for implementation of a feature—based on incorrect assumptions made previously. In that case, user feedback is the key for adapting requirements and improving the feature. This holds true primarily for minor changes in the user interface.

> **Observation 26:** Practitioners predominantly rely on user feedback for requirements verification instead of validation.

Change of User Feedback Over Time

In Figure 5.8, we visualize the practitioners' answers regarding sub-RQ 7.2. In 11 interviews, no answers were provided.

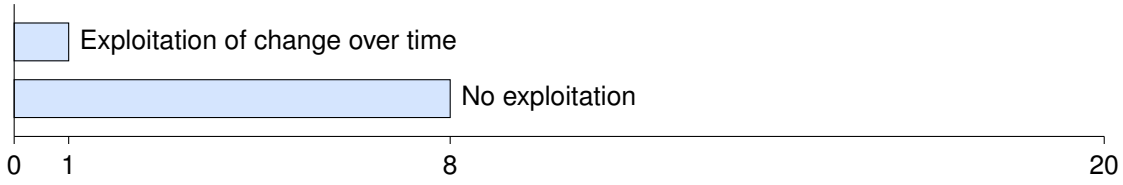


Figure 5.8: Number of interviews in which practitioners addressed change of user feedback over time. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

Only one practitioner reported to actively *exploit* changes in user feedback. However, this ability is highly correlated to the fact that they are using the data to understand and optimize the business value that the application is offering to its users.

Eight practitioners reported that they do *not exploit* changes in user feedback, while three practitioners elaborated that they are generally able to do that kind of investigations but have not yet done so. One practitioner stated that they have not yet encountered a situation in which an analysis over time would have been beneficial but sees that it could be useful in case of major changes in the user interfaces. Another practitioner shared similar thoughts and added that one always needs to consider the cost effectiveness, as this kind of analysis is something that could require major effort but yield only minor benefit. While stating that their tool probably allows for such a comparison over time, yet another practitioner indicated that they already struggle to cope with most of the basic functionality offered by the tool.

Out of those same eight practitioners, five responded that they lack the technical ability to exploit user feedback over time, but their general impression is that they would like to have such a functionality in place. For example, one practitioner supposed coarse-grained feedback at an early stage, while, after some iterations, they expected more fine-grained feedback to be provided by the user. An increase of the amount of user feedback may support understanding urgent feedback and help to prioritize it. This enables the detection of user feedback change events.

> **Observation 27:** Practitioners do not rely on exploitation of changes over time when utilizing user feedback, however, they show interest.

Combination of User Feedback Types

In Figure 5.9, we visualize the practitioners' answers regarding sub-RQ 7.1. In 12 interviews, no answers were provided.

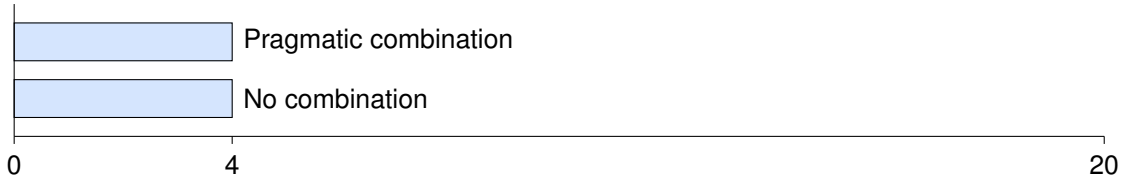


Figure 5.9: Number of interviews in which practitioners addressed combination of user feedback types. Figure adopted from Johanssen *et al.* [156] (© 2019 IEEE).

Four practitioners described *pragmatic combination* practices regarding user feedback. One practitioner reported to combine user profiles with written reviews, such as ones that originate from emails. Another practitioner stated that, while user feedback types are usually handled separately, different types of internal data can be combined. However, it is not being used for feature development. Two other practitioners emphasized that their starting point for the combination of different types of user feedback would be explicit feedback: if they observe peaks in email responses, they initiate a deeper investigation and include other user feedback sources.

Four practitioners indicated that they do *not combine* different user feedback types during development. However, while one practitioner did not see a benefit herein, two are interested in the ability to combine user feedback. Reasons why they do not combine feedback sources are a low complexity level of the application and limited availability of feedback.

> **Observation 28:** Most practitioners utilize user feedback individually, however, there is a trend toward its combination.

5.2 Discussion of Results

This section discusses results from Section 5.1 by combining practitioners' answers among RQs and relating them to literature. We derive recommendations, in which we reflect on either promising practices that are applied by a few practitioners but should be investigated in more detail, or challenges and untapped potential that were addressed in answers from multiple practitioners. The recommendations guide future work toward continuous capture and utilization of user feedback.

5.2.1 Continuous User Feedback Capture

More than a quarter of the practitioners rely on colleagues and team members in close proximity to retrieve an initial set of user feedback (Figure 5.5). While reports identified gaps in the requirements communication between departments [163], we found that the practitioners in our study drew positive conclusions regarding collaboration. They highlighted the benefits of direct contact that result in honest and instant responses. Fellow team members are more likely to provide detailed and structured feedback, as they have an intrinsic motivation to contribute to producing and maintaining high-quality software. Feedback from colleagues from outside their project is particularly welcomed, too, as they provide new perspectives on the feature under development. The close proximity to team members allows for frequent explicit user feedback capture with almost no delay. This combines the benefits of reported explicit user feedback with the requirements for CSE. For example, Facebook follows an approach in which their engineers act as an internal user [102].

> **Recommendation 1:** Continuous user feedback capture from internal sources should be expanded, as it provides honest responses and leads to diversified requirements.

The recommendation is an extension to the theory that testing with five users in multiple, consecutive, yet small experiments is most efficient to identify problems regarding requirements such as usability [219]. However, this is clearly insufficient:

Feedback must also be captured from external sources, as the developers and colleagues are not the domain specialists and relying on a selected user group over a long period increases the chances of bias [116]. A challenge remains in the transition from qualitative, low-quantity feedback from the limited scope of "users" in close proximity toward the high-quantity capture of user feedback from a large audience. This may be further refined in two transitions: From small developer groups to small user groups, and from small user groups to larger user groups.

The challenge becomes apparent in the light of Figure 5.1, which reveals that the majority of user feedback is explicit. The practitioners praised the value of face-to-face interactions and the richness of explicit user feedback. It comes as no surprise that explicit user feedback is preferred, as it allows practitioners to capture expressive insights with a high level of detail [116]. This is further emphasized by the high number of manual capture practices (Figure 5.4). Even during rapid development practices, practitioners rely heavily on manual approaches to capture user feedback, such as collecting user feedback from emails, questionnaires, or video recordings. However, as one practitioner pointed out, such user feedback capture and processing result in a non-agile procedure that slows down overall development and puts the feature's quality at risk. This is also reflected in the frequency with which practitioners capture user feedback. As depicted in Figure 5.3, only one quarter of practitioners continuously capture user feedback, while more than half rely on event-based and periodic approaches.

Reducing manual steps in favor of (semi-)automated processes may be one way of dealing with this issue. Specifically, this could be achieved by extending tool support, which has already been adopted by the requirements engineering community. For example, there are approaches that enable continuous context-aware user feedback [88, 196]. In addition, with respect to the combination of types of user feedback, FAME is an approach that combines explicit user feedback and usage monitoring to continuously elicit requirements [231].

> **Recommendation 2:** Continuous user feedback capture should adapt and extend tool support, e. g., by increasing the use of automatization for processing, to ensure high-quality and expressive requirements in a timely manner.

5.2.2 Continuous User Feedback Utilization

Practitioners reported the utilization of different types of user feedback, while their process on how to derive high-level and actionable insights is unsystematic: On the one hand, as visualized in Figure 5.2, many practitioners stated to relate user feedback to the entire application, leaving the feedback scattered across multiple features and thereby reducing its value for individual requirements. On the other hand, multiple practitioners indicated that they strive for a feature relationship while captur-

ing user feedback, such as through directly asking the user, or relying on user profiles and feature isolation. When elaborating on the tool support, one practitioner stated that commercially available tools are not fully adopted by practitioners. Following the terminology defined by Davis [75], we assess the practitioners' perceived ease of use of such approaches as low, which may be why many practitioners refrain from applying them in practice. It remains a challenge for developers to relate captured user feedback to a feature under development.

The Quamoco approach addresses the systematic mapping of quality metrics and abstract quality criteria [308], but only recently usage data-driven approaches are proposed [112]. Durán-Sáez *et al.* approached the challenge of relating requirements to fine-grained user events by logging them to derive implicit user feedback for various criteria [85]. For explicit user feedback, Guzman and Maalej developed a natural language processing technique that identifies an application feature from reviews and that allows attaching sentiments to it [133].

The users' context may provide additional support when relating user feedback to a feature under development. Knauss acknowledged that there is no specific definition of context in requirements engineering [171], which may explain the low number of responses in Figure 5.6. However, she went on to say that context is important to consider for requirements elicitation [171]. For example, increasing the availability of context information as well as the combination of explicit and implicit user feedback could further contribute to the extraction of *tacit requirements* [118], a challenge that developers face when users assume that their needs are obvious [51]. Generally, as one practitioner remarked, trivial context information, such as data points in the form of time stamps, already can significantly contribute to an increase in the value of user feedback.

> **Recommendation 3:** Continuous user feedback utilization should offer a lightweight concept for creating reference points to relate user feedback to requirements at hand.

User feedback is necessary for both the validation and verification of requirements [247]. Requirements validation is concerned with making sure that the specified requirements correctly represent the users' needs, i. e., that the elicited requirements are correct and complete. Requirements verification ensures that the design and implementation of a requirement correctly represent its specification. Agile development processes reportedly have a stronger focus on requirements validation than on their verification [253].

From the practitioners answers we cannot derive a unique reason why they capture user feedback, as many provided more than one reason (Figure 5.7). The answers for the two prevalent codings, i. e., planning and supporting, suggest that practitioners tend to capture user feedback as a means for requirements verification.

They reported to employ the user feedback to plan and to iterate on previous decisions, which contains a re-prioritization of whether to remove an existing feature or not, pointing toward usage of user feedback for requirements validation. According to them, these are high-level decisions, which is also reflected in the responses for Figure 5.2. Many practitioners, however, stated to apply implicit user feedback to guide support inquiries and work on bug fixes, i. e., they use implicit user feedback with a focus on requirements verification. Overall, the utilization of user feedback for a systematic validation of requirements is sparsely applied by practitioners, leaving opportunities unused. In particular the use for quality requirements was only addressed by a few practitioners. This, however, is a viable concept as it has been shown by Groen *et al.* that users are mentioning quality requirements, such as usability and reliability, when providing explicit feedback [127].

Notably, one practitioner mentioned work on an application for which neither a customer nor users yet exist, which made it difficult to elicit concrete requirements; they develop a software for which a demand still needs to be created. While they defined the application's functionality, their vision as to how it was to be presented to user was still vague. This meant that their software development was driven by understanding attractive requirements as opposed to must-be or one-dimensional requirements as defined in the Kano model [270]. As a consequence, the practitioners imagined a potential user, put themselves in their role, and used the application to then derive requirements for the feature based on their own feedback.

> **Recommendation 4:** Continuous user feedback utilization should enable the understanding of user feedback as a means to improve existing and to explore new requirements.

Fewer than half of the practitioners provided an answer to the question as to whether they address change of user feedback over time or its combination (Figure 5.8 and Figure 5.9). In fact, only one and four practitioners, respectively, reported that they have a working approach in place. This reduces the validity of assumptions derived from their answers; however, it strengthens a common theme among practitioners' answers: While many of them respond to be generally able to utilize user feedback, they are not making use of this possibility. We further observe this in answers regarding the capture of context (Figure 5.6) and the combination of user feedback types (Figure 5.1).

Expanding the involvement of the users during feedback utilization may help to improve this aspect. Stade *et al.* reported how tool support helped a software company to increase the communication with users [294]. Gómez *et al.* described a next generation architecture for app stores that enables more actionable feedback from users [121]. Maalej *et al.* predicted that *"[c]ommunication with users will move from being unidirectional [...] to being bidirectional [...]"* [197].

This vision is reflected in a few answers by practitioners who indicated that they are moving toward a dialog-based utilization of user feedback. Multiple practitioners provided concrete ideas. For example, regardless of the user feedback sources (Figure 5.5), practitioners stated that the ongoing and active exchange between developers and users has shown to be as successful in practice. One practitioner emphasized the value of retaining contact with close and active users, while actively informing them about new feature increments. This increases the possibility to retrieve more feedback and serves as a way to validate the changes that were triggered by a users' previous feedback, ensuring that the changes applied are in line with their needs. Similarly, another practitioner described a dedicated checkbox allowing users to request updates on the latest changes—in case they provided feedback.

With respect to Figure 5.7 and the goal to utilize user feedback for feature improvement, one practitioner envisioned a functionality that allows to define a mechanism that triggers as soon as a user starts interacting with a feature. This mechanism should enable developers to initiate a well-defined scenario that guides the users throughout their exploration of a new feature increment and to observe them while doing so. This would help the practitioners to pinpoint the users' attention to a detail on which they would like to have feedback.

> **Recommendation 5:** Continuous user feedback utilization should cover interactive practices to explore user feedback and increase the developer–user communication to exploit unused potential for enhancing the quality of requirements.

5.3 Related Studies

In this section, we list empirical studies that addressed how practitioners deal with user feedback and report results similar to ours.

Heiskari and Lehtola conducted a case study to investigate the state of user involvement in practice [138]. They found explicit user feedback as the most common type of insight for development—which is supported by our results in the Section 5.1.1.1. They identified a need to involve users in a more organized way than in the state-of-the-practice and to establish defined user feedback processes. This is reflected in the number of manual capture procedures described in Section 5.1.2.2.

Ko *et al.* performed a case study to identify constraints on capturing and utilizing post-deployment user feedback [172]. They found the heterogeneity of intended use cases of an application as the major constraint for the utilization of user feedback, as it makes prioritization difficult. Similar to this finding, the practitioners in our study noted that more context about the user feedback, such as user profile in Section 5.1.2.4, is needed to support the utilization of the feedback.

Stade *et al.* reported on a case study and online survey on how software compa-

nies elicit end-user feedback [293]. Their research questions partly overlap with our research questions: They found a need for more context information, which again corresponds to our results in Section 5.1.2.4. They identified feedback channels for user feedback such as hotlines or phone calls, email, and meetings being used the most. This confirms our results in Section 5.1.1.1. While Stade *et al.* provide more precise results in quantitative terms than ours for feedback channels and types, they do not address implicit user feedback through monitored usage data.

Olsson and Bosch investigated post-deployment data usage in embedded systems during three case studies [144]. Post-deployment data covers diagnostic data that supports troubleshooting activities such as bug fixing, but also operational data necessary to understand the usage (or non-usage) of individual features. They found that post-deployment product data was neither used to acquire insights on individual feature usage, nor was it employed to derive user patterns related to specific system functionality and to innovate new functionality [144]. In our interview study, we did not focus on a certain type of software, such as embedded systems. Nonetheless, our results in Section 5.1.3.1 support the predominant use of user feedback for planning and bug fixing, rather than feature improvement in the sense of innovation. In addition, our results also support their finding that practitioners do not investigate on user patterns, which could be derived from user feedback over time, as it becomes evident from Section 5.1.3.2.

Pagano and Bruegge reported on a case study with five companies in which they investigated the user involvement and the way how professionals deal with *"post-deployment end-user feedback"* [236]. They stated that the utilization of user feedback is mainly a manual process, which is reflected in our results in Section 5.1.2.2. Similar to our recommendations, they conclude that tool support is required to make use of the captured utilize user feedback for other practices, such as for requirements engineering [236].

Chapter 6

Usage Knowledge in Continuous Software Engineering

> *"Only objective knowledge is criticizable: subjective knowledge becomes criticizable only when it becomes objective. And it becomes objective only when we **say** what we think; and even more so when we **write** it down, or **print** it."*
>
> — KARL L. POPPER [249]

Following the focus on user feedback in the previous chapter, we want to investigate the possibilities of usage knowledge, which reflects a class of user feedback that provides both a relation to a feature and the need for action as described in Section 2.1.3, in the context of CSE. CSE offers new opportunities for the management and utilization of such usage knowledge. As introduced as part of the CURES project description in Section 3.2, our overall goal is the integration of different knowledge types into CSE to support software evolution [153, 154]. Therefore, we presented the so-called CURES vision in Figure 3.1.

We were interested in understanding the practitioners' thoughts on the proposed vision. Therefore, based on the case study described in Section 3.3, we investigated on the research questions introduced in Section 3.3.1.3 that address Knowledge Question 3. We report on the results in Section 6.1: First, we addressed their overall impression and opinion on the feasibility of the CURES vision in order to study their general attitude. Second, considering they would apply the CURES vision in their company, we asked the practitioners for their perceived benefits and obstacles. Third, we studied their feedback on how they would extend the vision, as well as potential additions which would require fundamental changes. Based on the practitioners' responses, we evolved the CURES vision and derived an improved version which we describe in Section 6.2.

6.1 Results on How Usage Knowledge Supports CSE

For each RQ, we provide a summary followed by a more detailed analysis: With respect to RQ 8, we analyzed the practitioners' attitude throughout the interview to derive their overall impression and their opinion on the overall feasibility, both grouped by positive, neutral, and negative responses. For RQs 9 to 12, we analyzed the occurrence of the codes *usage knowledge focus*, *decision knowledge focus*, and *other focus* in relation to the answers to a research question. In this dissertation, we focus solely on the reporting of the usage knowledge codes; however, we describe the coding process in more detail in Section 3.3.4.

During multiple interviews, we noticed that benefits were refined in a later question or used as an obstacle when viewing it from a different perspective. Further, answers to RQ 11 and RQ 12 were often very similar and dependent on how we phrased the questions during the semi-structured interview. We relied on a fine-grained distinction since the questions put different emphases: While the extensions refer to the proposed CURES vision (Section 3.2) and do not assume major changes, practitioners' responses regarding additions tend to result in fundamental changes to the CURES vision. In Figure 6.1, we visualize an overview of the responses separated by the respective RQ. A more detailed analysis of the responses is provided in Figures 6.4, 6.5, 6.6, and 6.7, in which we summarize topics of responses in case they were mentioned by at least two practitioners—which may also included other codes; therefore, these numbers can deviate from the ones presented in Figure 6.1.

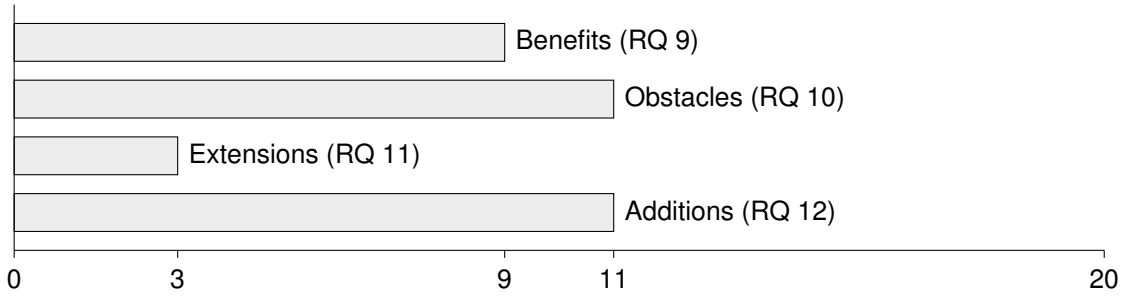


Figure 6.1: Number of interviews in which practitioners reflected on usage knowledge in the CURES vision, separated by RQs. Adapted from Johanssen *et al.* [158].

6.1.1 Attitude toward the CURES Vision

We asked the practitioners to provide their overall impression and opinion on the feasibility of the CURES vision to understand their attitude. We classified their response as *positive* if it clearly verbalized their support for one of the aspects. *Neutral* responses reflect responses that do not provide a definite point of view, and *negative* responses a substantial concern. We visualize the results in Figures 6.2 and 6.3.

Summary RQ 8: The majority of practitioners reacts positive toward the integration of knowledge types into CSE. Regarding the practitioners' overall impression, 1 negative, 3 neutral, and 15 positive responses were collected. For the overall feasibility, 7 practitioners provide positive feedback, while 11 state a neutral- and 2 a negative opinion.

In 15 out of 20 interviews, the practitioners noted a positive impression by describing the ideas as *"beneficial"* or *"exciting"*, as well as emphasizing the advantages of a certain idea or aspect. Many of the practitioners acknowledged different aspects of the CURES vision, such as the documentation aspects, as important and continued to refine their responses when discussing RQ 9. During four interviews, the practitioners did not clearly take a stance for or against the proposed vision, however, they showed interest and did not articulate negative impressions. While generally being interested in the vision, one interviewee determined it impossible for their project at hand, rendering the vision only beneficial for specific projects; a closer elaboration on their reasoning is provided as part of the obstacles in RQ 10, where their stance is summarized under applicability, i. e., finding themselves in a state in which multiple software versions are delivered to users. In Figure 6.2, we visualize the responses.

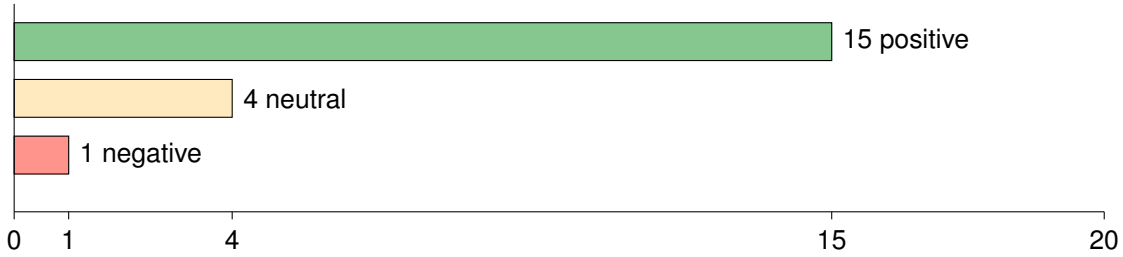


Figure 6.2: Number of interviews in which practitioners expressed their overall impression on the CURES vision. Adapted from Johanssen *et al.* [158].

With respect to the practitioners' attitude toward the overall feasibility of the CURES vision, we noted that the practitioners provided feedback dependent on their roles in the project; for instance, developers' responses mainly addressed technical aspects. Out of the 20 interviews, we received seven positive responses, expressing that they can imagine the implementation of such a concept. Some of them noted that many elements of the vision have already been addressed by individual, detached systems, but given a vast amount of effort, they consider the vision to be feasible. With eleven interviews, the majority of practitioners took a neutral view on the vision's feasibility by not clearly stating their opinion; some practitioners see challenges in the applicability in real-world scenarios, since the vision presented in Section 3.2 reflects a simplified scenario. A more detailed analysis of their responses is summarized under the obstacle of feasibility in the *obstacles* discussion

of Section 6.1.3. Two practitioners expressed a negative opinion regarding the overall feasibility of the CURES vision. They grounded their doubts in technical- and process-oriented challenges. In Figure 6.3, we visualize the responses.

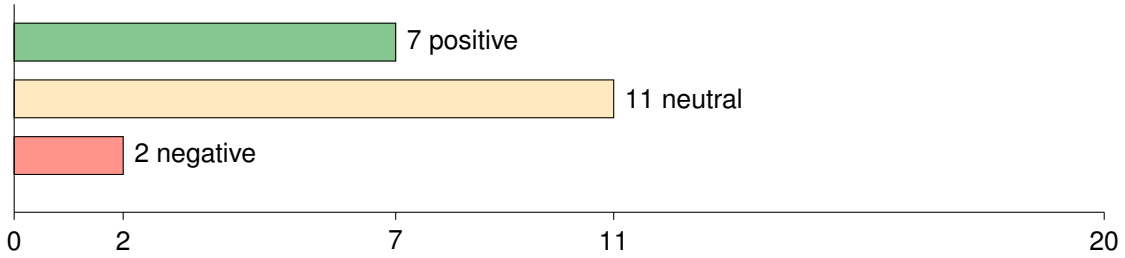


Figure 6.3: Number of interviews in which practitioners expressed their opinion on the overall feasibility on the CURES vision. Adapted from Johanssen *et al.* [158].

6.1.2 Benefits of the CURES Vision

After an introduction to the CURES vision, we asked practitioners about benefits considering they would apply it in their company. We refined our questions by asking the practitioners to relate their answers to the components of the CURES vision, such as the *Dashboard* as depicted in Figure 3.1, in case they did not know where to start reporting. We consolidated their responses and found five major benefits that address usage knowledge.

Summary RQ 9: From the practitioners' responses, we derived five benefits that address usage knowledge: Accountability, traceability, parallel feedback, automation, and flexibility. We identified 12 answers that relate to usage knowledge, while parallel feedback was the most popular benefit. We visualize the distribution of benefits across interviews in Figure 6.4.

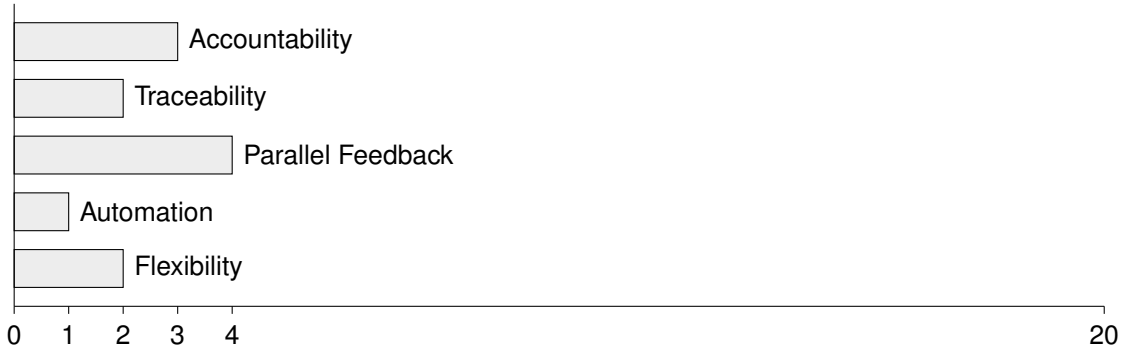


Figure 6.4: Number of interviews in which practitioners mentioned an individual benefit with respect to the CURES vision.

Accountability

Three practitioners stressed accountability benefits when combining usage and decision knowledge during the development process: the user feedback supports the product management to make decisions regarding the next step of the product development. Overall, the *Dashboard* and the *Knowledge Repository* to integrate usage and decision knowledge, as introduced on Figure 3.1, are understood as a central point of interaction and discussion.

> **Observation 29:** Practitioners highlight that the explicit and structured way of collecting and visualizing usage knowledge improves transparency, comprehensibility, and replicability.

Traceability

Two practitioners emphasized the benefits of feature-based separation of concerns: it allows to allocate knowledge directly to feature branches, which simplifies the process of retrieving documented knowledge afterwards. Developers can track down knowledge right where it is created. In particular, traceability links to issue entries and deployed releases are automatically established. This enables easily review of the decisions made during particular software increments with respect to the collected usage knowledge—a process that previously involved manual efforts. Three practitioners highlighted that interweaving different components, such as the *Knowledge Repository* with the CSE *Infrastructure* (see Figure 3.1), increases traceability, in particular with respect to usage knowledge.

> **Observation 30:** Feature branches as a main reference point simplify the traceability of usage knowledge.

Parallel Feedback

The practitioners consistently responded that they perceive an environment in which they can evaluate two different feature implementations as highly beneficial for their development process, as it allows to contact users directly, which increases the chances to receive instant feedback. In this context, they emphasized the possibility to simply branch and merge the feature branch that exhibits the better performance, a welcomed support in their day-to-day work. Two practitioners highlighted that the vision shows similarities to A/B testing and showed their interest.

> **Observation 31:** Practitioners point out benefits in being able to contact users directly and receive instant feedback.

Automation

The practitioners mentioned benefits in the fact that the components support developers through semi-automated processes. For example, user feedback on an old release may be automatically mapped to the corresponding, but deprecated proposal. This allows the current development activities to focus only on relevant feedback and prevent the developers from distracting information. In addition, automating this process supports consistency and comparability of test results.

> **Observation 32:** Automation supports practitioners in synchronizing knowledge artifacts and maintain a focus on current activities.

Flexibility

Two practitioners highlighted that CSE elements such as continuous delivery ease the process of deciding from whom usage feedback is to be collected. In particular, they noted that the delivery of release to collect feedback from either internal or external sources is particularly simplified, resulting in a flexible work environment.

> **Observation 33:** The CURES vision enables a fine-grained usage knowledge collection and assessment which can be adjusted to developers' needs.

6.1.3 Obstacles of the CURES Vision

We asked the practitioners for obstacles to the implementation of the CURES vision within their project which go beyond their opinion on the overall feasibility addressed in RQ 8. We did not further specify the interpretation of an *implementation*, which is why practitioners referred to various areas, reaching from development- to project-related aspects. We consolidated their responses and found four major obstacles that address usage knowledge.

> **Summary RQ 10:** From the practitioners' responses, we derived four obstacles that address usage knowledge: User groups, applicability, usability, and privacy. We identified 15 answers that relate to usage knowledge, while user groups was the most popular obstacle. We visualize the distribution of obstacles across interviews in Figure 6.5.

User Groups

Practitioners highlighted that the user groups that provide user feedback tend to be internal development teams, not the targeted end user groups, which may make it

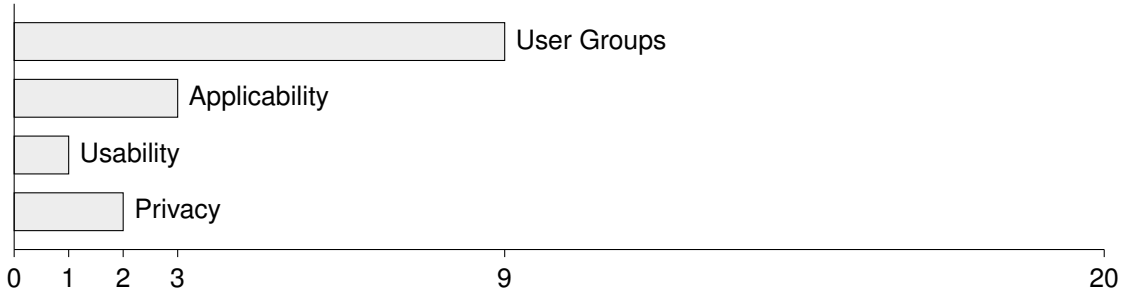


Figure 6.5: Number of interviews in which practitioners mentioned an individual obstacle with respect to the CURES vision.

difficult to use the received feedback as a representative baseline for decisions. This relates to Observation 24 that we derived during the focus on user feedback. They continued to highlight that—to correctly distinguish between two feature implementations—it requires many users to retrieve statistically significant results; this is barely reachable in such an environment. Two practitioners specified their concerns of finding users that provide usable feedback: First, it sometimes requires effort to understand the changes which reduces the users' acceptance to provide feedback. Second, the process of selecting users based on their skills is an important step, since it allows to ensure valuable feedback; however, this possibility is not provided with the original CURES vision as presented in Section 3.2.

> **Observation 34:** Practitioners prefer a system that can be used in a day-to-day workflow that is closely related to users, similar to an environment in which a few of their colleagues can briefly be consulted to receive feedback on a new software increment.

Applicability

Three practitioners noted a high risk in providing different features to their user audience, as it would directly interfere with their business processes. One practitioner reported that collecting usage data is a business-critical part of the product, which makes different versions of a software feature difficult, if not impossible to justify and bill. Two practitioners argued that the CURES vision is incompatible with their way of developing and delivering software due to safety-critical aspects. Two practitioners mentioned technical problems in delivering multiple feature versions as demonstrated in the CURES vision. While they think that the visions can be implemented in general, it would require them to apply major efforts and adjustments in their development process to implement such changes. Finally, the practitioners mentioned a high coupling to server-side information which makes the applicability of the vision problematic. In particular, different staging environments that are used

during the development make the supply of different client-side versions challenging. Furthermore, even if these challenges would be resolved, repeating the process continuously would introduce new challenges.

> **Observation 35:** Practitioners are concerned about the effects on run-time aspects when creating different versions of a feature at the same time.

Usability

Some practitioners wondered why a developer should use such a system, especially when they already have to deal with other, even more complex systems. In addition, the practitioners articulated doubts whether the correct stakeholders are addressed with the current version of the vision: this is the only time practitioners noted not enough roles as an obstacle. Two practitioners mentioned rather practical obstacles: while one practitioner noted that the proposed vision would solve a problem that they are currently not confronted with, another practitioner predicted upcoming problems in managing a new system in particular with respect to the dashboard component, as it needs to be integrated into user management systems and requires a variety of configurations.

> **Observation 36:** The integration of usage knowledge into CSE must be easy to use to be adapted by stakeholders.

Privacy

The practitioners pointed out that the data that gets collected provides detailed insights into the user base. This may contain sensible information, which may be difficult to protect when processed and presented in other systems, such as the *Knowledge Repository* or *Dashboard* introduced in Figure 3.1. One practitioner stated that this obstacle can only be overcome if the development is clearly separated from production, which, however, interferes with the idea of continuous software engineering—following the quote *"You build it, you run it"* [229] the practitioner indicated that this obstacle needs to be addressed.

> **Observation 37:** Practitioners have privacy concerns when applying the CURES vision with real users. Therefore, the practitioners suggest applying parts of the CURES vision in development environments only.

6.1.4 Extensions to the CURES Vision

We asked the practitioners for extensions to the presented CURES vision. In particular, we required them to relate their answers to their perceived benefits and how

they could be strengthened. Likewise, we asked the practitioners to provide extensions to overcome obstacles and improve the overall feasibility of the CURES vision. Extensions are classified to be short-term changes to the CURES vision that do not require a major change and can be easily integrated into the existing concepts. We consolidated their responses and established four major extensions that address usage knowledge.

Summary RQ 11: From the practitioners' responses, we derived four extensions that address usage knowledge: Automation, roles, run-time, and human and processes. We identified six answers that relate to usage knowledge, while no noticeable peak for a particular extension. We visualize the distribution of extensions across interviews in Figure 6.5.

Figure 6.6: Number of interviews in which practitioners mentioned an individual extension with respect to the CURES vision.

Automation

The practitioners shared a common idea for an extension to automate every aspect of the system. One practitioner formulated automation steps based on certain actions; for example, as soon as a feature branch is created, other processes such as creating a representation of this very feature branch in the *Knowledge Repository* should be triggered. They mentioned that it would be sufficient to have a basic support for this functionality, such as following naming conventions, in order to make it work. Another practitioner stated that automation needs to be implemented differently for individual stakeholders and that a lowest common denominator should be found; otherwise, there are multiple things that can make such a system fail. As soon as one aspect that could be automated requires effort, such a system is difficult to establish; one practitioner justified the need for automation with the fact that—if there is just one aspect that needs to be done manually—the data will turn inconsistent over time. Two practitioners highlighted that automation allows them to see the relevant information at the time it is needed and enables to drill-down on information at hand.

> **Observation 38:** With respect to the CURES vision presented in Figure 3.1, practitioners request an additional component between the *Knowledge Repository* and the CSE *Infrastructure* that is in charge of performing various automation steps, such as showing relevant usage knowledge when available or synchronizing both components in case a change occurs.

Roles

Multiple practitioners stressed the need for extending the CURES vision to more roles than developer and user. Four practitioners stated the need for adding capabilities for a project manager, i. e., a management role. According to their statements, these roles could benefit from the dashboard by collecting information required for resource planning and estimation. At the same time, they could use it to be updated about latest developments, such as the delivery of certain software versions to users, or the availability of features in a feature branch. Then, the dashboard serves as an access point to information which previously required other tools, such as a git client, which some project manager might refrain from using. Two practitioners highlighted different roles in the target audience, i. e., the users, and suggested to distinguish between internal users, such as developers, and external users. Another practitioner justified the need for extended roles in being able to comply with privacy aspects: more roles allow for a more fine-grained selection of who is allowed to see which information.

> **Observation 39:** Practitioners stress the need for extending the CURES vision to more roles than developer and user to control access to usage knowledge.

Run-Time

With a focus on technical aspects, the practitioners expressed an interest in being provided with monitoring data from technical logs. This would allow them to understand the features that are used by the users and control server-side capacities accordingly; based on metrics shown in the dashboard, indications for roll-out or release dates could be inferred. One practitioner indicated practical extensions that could be easily added to the current implementation: they would like to have an easy way to access older versions of a release, which could be integrated into the dashboard. Another practitioner requested the ability to see more dependencies to other software and physical devices to adjust the release of a feature at hand.

> **Observation 40:** Practitioners are interested in run-time knowledge that goes beyond usage knowledge to control operational aspects.

Human and Processes

The practitioners highlighted that that by applying the CURES vision, involved individuals need to rethink existing processes. The practitioners acknowledged that the human factor needs to be addressed for a successful implementation. To convince individuals of processes, practitioners suggested to provide support such as acknowledging that—initially—more effort is required to make the process work. Likewise, they encouraged the integration of process additions, such as making pull requests dependent on the availability of a certain type of knowledge, to foster the capturing of knowledge and to promote the usage of the system. One practitioner noted potential of the CURES vision in making aspects of the process visible, which might have been undiscovered before.

> **Observation 41:** The integration of usage knowledge in CSE needs to be acknowledged by individuals and processes in order to be successful.

6.1.5 Additions to the CURES Vision

We asked the practitioners to list major additions, which they would expect from a system to capture, maintain, and explore knowledge during CSE. We strived for collecting long-term additions to the CURES vision that go beyond the presented vision and that require a major refinement of the concept. We consolidated their responses and established five major additions that address usage knowledge.

> **Summary RQ 12:** From the practitioners' responses, we derived five extensions that address usage knowledge: Integration, experimentation, interaction and reaction, granularity, and developer focus. We identified eleven answers that relate to usage knowledge, while experimentation was the most popular addition. We visualize the distribution of additions across interviews in Figure 6.7.

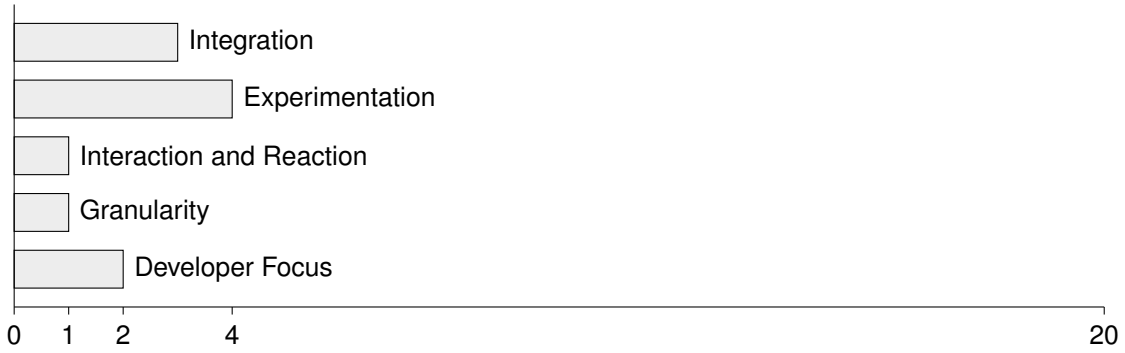


Figure 6.7: Number of interviews in which practitioners mentioned an individual addition with respect to the CURES vision.

Integration

The integration with tools available in a company's development process is stated most prominently by the practitioners: Two practitioners noted support for external usage knowledge systems, i. e., to receive a greater variety of qualitative and quantitative feedback. Others requested support for interfaces to systems that are close to hardware, i. e., to be able to read and write data via markdown or plain text, to collect more information. Overall, the integration should increase the stakeholders' acceptance by focusing on usability and efficiency.

> **Observation 42:** Future concepts of the integration of usage knowledge into CSE need to offer a high tool integration.

Experimentation

Five practitioners expressed specific feature additions: One technical leader described scenarios in which users are required to find themselves in a particular setting to test a new feature addition; this setting may be given through a particular physical surrounding or a selection of parameters within the application. The practitioner imagined to formalize such scenarios to be able to put users automatically into a setting in which they can use the feature. This, however, requires a high maturity of integration processes, which makes its applicability for small features less likely. One practitioner expressed the vision of defining experiments in a way that a continuous representation of results can be visualized, for example, by a numeric value. Another practitioner evolved this idea by being able to measure any kind of interaction with the user to derive valuable information for visualization in a dashboard. They suggested to observe time spans until a feature is understood by a user and try to derive phases in which the users find themselves—such as a learning phase, identification phase, orientation phase, searching phase, or productivity phase. Two practitioners focused their needs for additions in the interplay with users; they would like to address certain groups of users to specifically release a new feature to them. They proposed that users can make the decision themselves, whether they want to use and provide feedback for a specific feature version.

> **Observation 43:** Collecting usage knowledge and providing monitoring and feedback components provides the opportunity for experimentation in CSE.

Interaction and Reaction

Adding more interaction possibilities to a dashboard was perceived as beneficial by multiple practitioners: they would like to control released versions from within the dashboard. Similarly, they requested the possibility to connect and link existing

knowledge in order to create new knowledge. Apart from the interaction aspects, the practitioners expected the dashboard and its underlying knowledge repository to be reactive: for instance, the system should be able to send notifications to the stakeholders in case a previously defined threshold is reached. Likewise, relationships should be identified intelligently and presented to stakeholders by suggestions or warnings for interaction.

> **Observation 44:** Practitioners request to extend the functionality of the CURES dashboard beyond static representation of usage knowledge.

Granularity

The practitioners expressed the need for more layers to the feature representation in the knowledge repository, in order to differentiate between knowledge for low-level and high-level decisions. This would be similar to a differentiation made in project management, such as between *epics*, *user stories*, and *implementation tasks*, and be reflected in the granularity of the collected usage knowledge.

> **Observation 45:** Practitioners request major additions to the granularity of knowledge collected in the knowledge repository.

Developer Focus

Two practitioners elaborated on relying more on the developer to collect usage knowledge: Since developers know how to trigger a certain behavior of the system, they should be able to simulate a user and produce data which is treated separately from the data produced by actual end users. An additional component could enable the replay of a situation which allows them to better understand a problem. Introducing this developer focus and separating it from other, end user mechanisms for usage monitoring would allow a more rigorous, yet effective way of collecting usage knowledge, without inferring with privacy aspects.

> **Observation 46:** The CURES vision can benefit from providing the ability to let developers assess their own behavior.

6.2 Discussion of Results

From the practitioners' responses, we collected many insights of how usage knowledge can support them to improve the results of CSE. The assessment of RQ 8 in Section 6.1.1 shows that the majority of practitioners have a positive overall impression of the CURES vision. Regarding their opinion on its feasibility, neutral responses

dominated the practitioners' responses, followed by seven positive statements: As most of the neutral responses can be related to either no clear statement or general remark, we also conclude the feasibility aspect as accomplished.

We combined the practitioners' responses from RQ 9 to 12 and updated our original CURES vision presented in Figure 3.1—the result is sketched in Figure 6.8 and detailed in the remaining part of this section, combined with two examples of how the improvements could be instantiated.

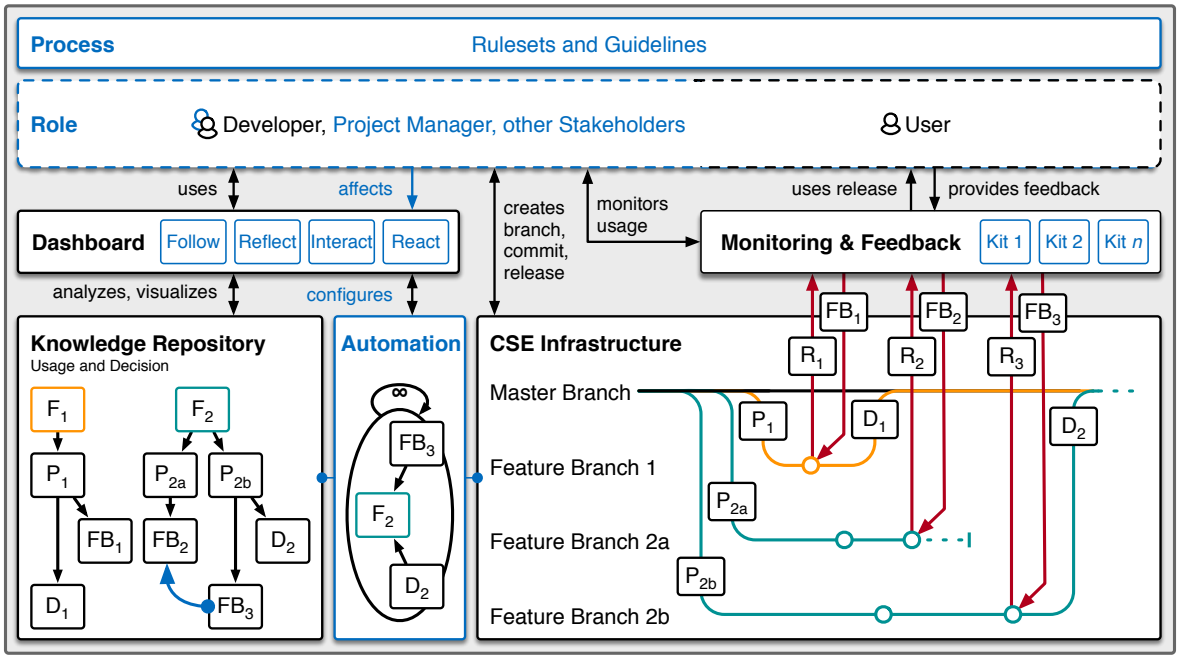


Figure 6.8: The improved version of the CURES vision refined on the basis of the interview results, adapted from Johanssen *et al.* [158].

With respect Figure 6.8, changes and additions to the version introduced in Figure 3.1 are highlighted in blue color: major improvements can be found in a new *Automation* component, the introduction of *Rulesets and Guidelines*, as well as the extended functionality of the *Dashboard* and *Monitoring and Feedback* component. Furthermore, the improved vision considers more types of *Stakeholders*. Finally, the *Knowledge Repository* received refinements and the CSE *Infrastructure* minor changes.

A major addition is represented by a new *Automation* component; multiple practitioners pointed out that the separation between the knowledge repository and the CSE infrastructure is too strict. While the original CURES vision intended knowledge elements to be transmitted directly between these components, the new automation component makes it clear that this may be done by an additional component, which may also be implemented using external systems. We see a broad field of tasks that can be accomplished by the automation component. Following the practitioners' suggestions, it can perform a set of tasks as soon as an event is triggered, such as collecting all available information for a feature branch that has just been created, as

described as the automation extension; this could also mean to collect other knowledge from technical logs, as described as a run-time extension. The automation component can also be in charge of regularly checking the knowledge available in the knowledge repository and investigating whether there are deprecated knowledge elements or missing knowledge links, as described as the automation extension. The component may also responsible for preparing knowledge that could be used within the new extensions of the dashboard, as explained in the following.

> **Example 9:** Knowledge originates from different sources, i. e., is created within other platforms. For instance, the CSE infrastructure might be based on an online repository that is maintained independently from a server that hosts the knowledge repository or the monitoring and feedback component. A hook architecture can enable a connection with is required to achieve the goals of the automation component. Web hooks represent a simple way of connecting independent sources, since they usually authenticate only via a web address and a secret token; this make it easy to integrate them into already existing systems. Furthermore, they are event-based, which fits the requirements for the automation component: whenever a new commit is pushed to the code repository, the knowledge repository can receive a notification from the CSE infrastructure. Based on this, it can initiate the gathering of data. Likewise, whenever a user triggers a new feedback, this knowledge can be pushed into the knowledge repository and automatically related to a commit.

The automation component is further linked to the *Dashboard*, which was addressed by the practitioners as part of multiple responses. Initially, the dashboard's intention was to *consume* knowledge that is stored in the knowledge repository—now represented by the *Follow* and *Reflect* components in Figure 6.8. Following the practitioners' feedback, we found it important to be more specific about the possibilities the dashboard offers [154]: The *Interact* and *React* components make the dashboard more active, offering the possibility to inform stakeholders about relevant changes, as described as the interaction and reaction addition. A new aspect is given through the *affect* relationship from the role toward the dashboard; depending on *who* is using the dashboard, it changes its appearance and the way it works which addresses the obstacle of privacy.

Several practitioners addressed the limited set of *Roles* in the original version of the CURES vision, such as with respect to usability as an obstacle.

Therefore, we extended the scope of roles, considering any possible stakeholder with an interest in the product development and implementing the proposal for an extension formulated as roles. For instance, this may als be project managers or any other stakeholder. Furthermore, by fusing the user role into the other stakeholder roles, we acknowledge that our CURES vision is planned to be used in a

development-close environment, in which a developer is the first *user*, followed by other stakeholders, of which one of them might represent the actual end user. This removes uncertainty regarding the obstacles of user groups, i. e., the availability of end users, as well as their necessity to have a certain level of knowledge and to invest effort. Also, it increases the obstacle of applicability, since—if performed by a developer—business- and safety-critical areas of a software product can be tested in a controlled environment. Finally, having developers act as users represents another mitigation of the privacy obstacle, as they are interacting with the application while they know that they are being observed. While this might influence the feedback, it accommodates the request for an addition to be more developer-focused.

We improved the capabilities of the *Knowledge Repository* following the request of multiple practitioners. We enable links between knowledge elements, such as between $\texttt{FB}_2$ and $\texttt{FB}_3$, to increase the degree of knowledge granularity as described as suggestion for addition by practitioners.

Motivated by the benefit of parallel feedback, we enhanced the composition of the *Monitoring and Feedback* component. With the introduction of *Kit* components, we implemented the suggestion for addition in form of experimentation, as different modules for user understanding can be added to the monitoring and feedback component. In the case of a special knowledge need, a kit can be activated and used for collecting usage data; this might even be in form of an external service that offers specialized monitoring features, as proposed by practitioners as an addition following integration.

> **Example 10:** Stakeholders might have different interests about the usage knowledge that is collected. This need is reflected in the possibility to use kits. For example, one instance of a kit may allow stakeholders to observe and analyze the status of newly developed features increments: information whether a feature was started, completed, or canceled helps to understand the usability of a feature. Other kits can provide additional data, such as information about the users' behavior or their impression that was derived from a hardware or a software sensor. By combining these knowledge sources, the decision-making on the next steps for the feature increment can be supported or improved.

A minor addition lies in the introduction of a new *Process* layer, which implements the extension in form of human and processes, which are required to successfully implement the CURES vision. This may be reflected in the introduction of a basic naming convention to support the automation aspects of the CURES vision, as suggested by developers as an extension for automation.

Part III
Treatment Design

TREATMENT DESIGN is concerned with finding a way to approach a problem, while there can exist multiple answers [310]. In this part, we introduce our treatment for the problem context. On a high level, it is represented by a set of artifacts that we summarize under the umbrella of the **Cuu**$^{\text{SE}}$ framework.

The structure of the chapters is inspired by the three core components of the Blackboard pattern [52]: We present the **Cuu** workbench and feature crumbs as an extension for a blackboard, knowledge sources are reflected in so-called **Cuu** kits, and the ability to control a Blackboard is reflected in the **Cuu** workflow.

For each individual treatment artifact that is part of the **Cuu**$^{\text{SE}}$ framework, we applied the steps of the design cycle. In the following, we describe elicited requirements, followed by a conceptual description of the treatment component. We provide sketches of actual artifact designs that instantiate the concept as a means for validation. Furthermore, we provide a review of related treatments, which serves as an answer to knowledge questions and to closes the complete of treatment design cycles, i. e., by investigating other possible treatment for the problem in question.

Chapter 7

A Framework for Continuous User Understanding

> *"[The] use [of] fragments of many different kinds of data [...] seems to fit better the requirements of problem-solving, in which relatively weak and scattered information must be used to guide the exploration for a solution. [...] Metaphorically[..] we can think of a set of workers, all looking at the same blackboard: each is able to read everything that is on it, and to judge when he has something worthwhile to add to it. [...]"*
>
> — ALLEN NEWELL [214]

In this chapter, we describe the architecture and the main components that are part of the **CuuSE** framework, for which we rely on the metaphor of a blackboard system [66, 67, 221, 222]. We use the architectural design pattern as described by Buschmann *et al.* to outline the framework on a high level in Section 7.1: While there are differences in the characteristics of how a blackboard pattern works, its core benefits as well as the intended application domain, i. e., when there is no deterministic solution strategy, reflect the problem context of **CuuSE**. In Section 7.2, we describe a monitoring concept which forms an extension to the original blackboard pattern as it provides a frame to the input stream of raw data that is added by knowledge sources to the blackboard. In Section 7.3, we describe the blackboard's repository, to which we refer to as the workbench as it encompasses more functionality. It consists of two major components, i. e., a dashboard for usage knowledge visualization to enable access by developers, as well as a platform, which offers the environment in which the framework's components operate. In Section 7.3.5, we describe the designed artifacts that resulted from the concepts that were introduced in this chapter. They serve as the main treatment for validation in Chapter 12.

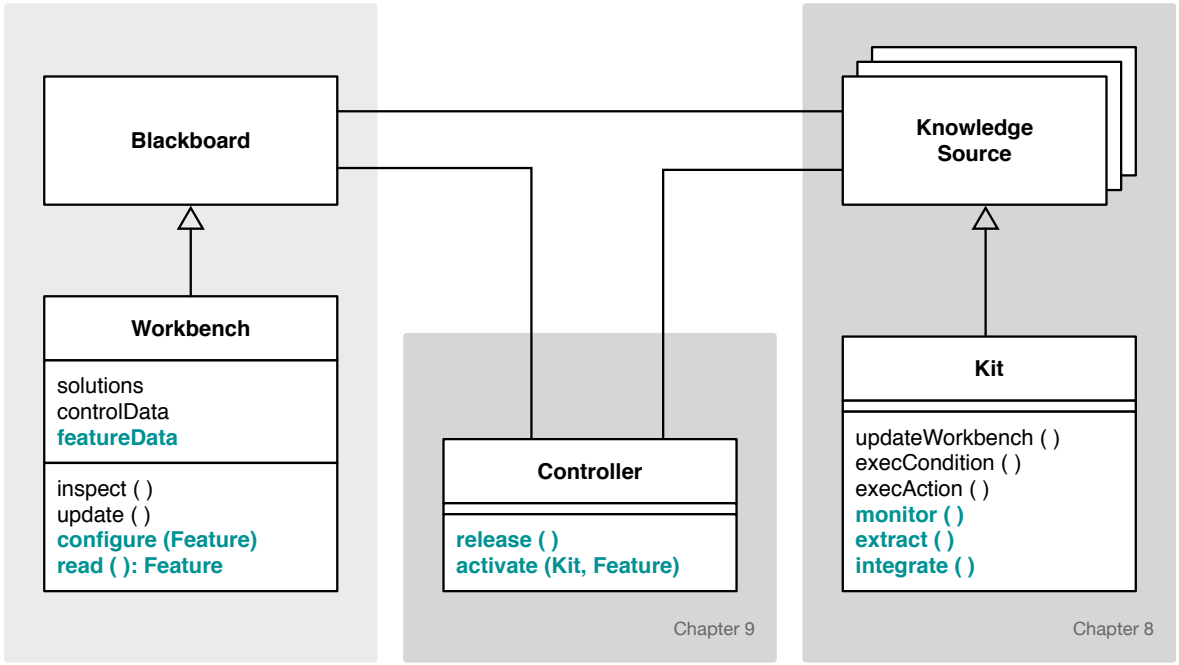


Figure 7.1: The **Cuu^{SE}** framework's core classes, i. e., workbench, knowledge source, and controller, base on the blackboard pattern as described by Buschmann *et al.* [52].

7.1 Architecture Design

The extraction of tacit knowledge represents a *non-deterministic* problem, as there does not exist a definite strategy that may be used to understand users: For such cases, Buschmann *et al.* propose to use a blackboard architectural pattern, in order to "transfer raw data into high-level data structures" [52]. In our problem context, a basic representation of the raw data are interaction events triggered by the user when navigating through the user interface. Following the usage taxonomy introduced in Section 2.1.3, several domain experts may be able to apply their expertise on an input stream of raw data, i. e., usage data—or on an already processed input stream of usage information—in order to process it toward a higher-level representation of usage knowledge. In general, to understand users and extract their tacit knowledge, it is not possible to define and repetitively apply a fixed sequence of activities; this requirement is reflected in the blackboard pattern. In Figure 7.1, we adapt the structure of a blackboard as described by Buschmann *et al.* [52] and extend the three core classes to describe the high-level architecture of the **Cuu^{SE}** framework; Bold font represents attributes and methods that were added or reimplemented.

The `Controller` remains similar to its counterpart in the blackboard pattern, however, its authority in the on-going event flow is reduced. To reflect the development process, the original loop method is replaced with `release`, which initiates the execution of the **Cuu^{SE}** framework. The release method `configures` the blackboard with the feature that is currently being developed. Likewise, it uses the

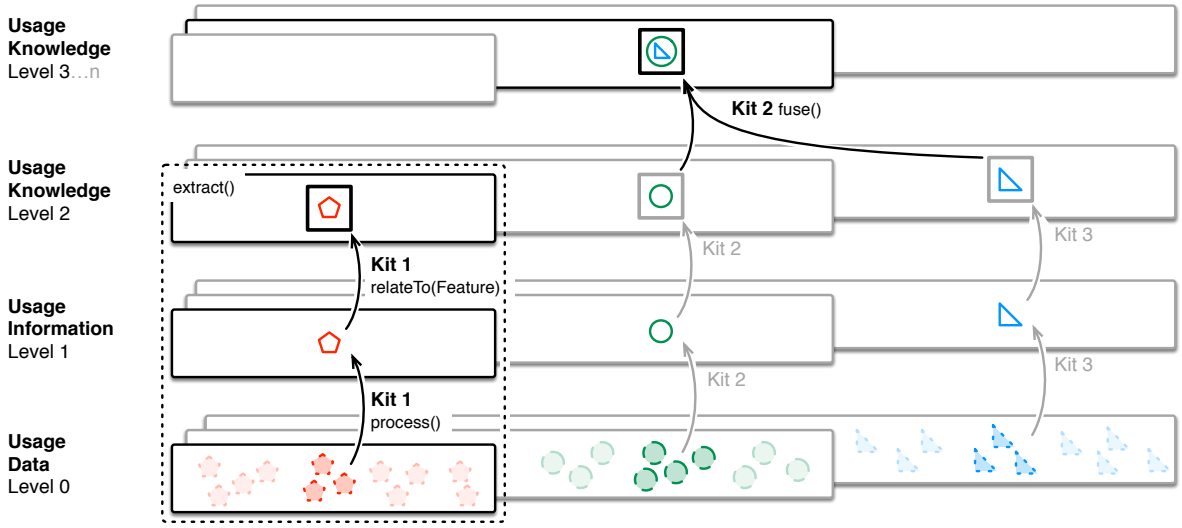


Figure 7.2: Informal object model of raw data transformation in **Cuu**^{SE} that shows how raw usage data is transformed into higher-level representations. The level-based structure of the figure follows the representation used by Lesser *et al.* [190].

`activate` method, which replaces the blackboard pattern's *nextSource* method, to identify and start relevant knowledge sources as well as to provide them with the same feature configuration. We define the structure of the `Feature` definition in Section 7.2.2. After they have been activated, the knowledge sources operate independently toward finding the best composition of usage knowledge provided available usage data. Therefore, future versions of the **Cuu**^{SE} framework may no longer require the control class, as the knowledge sources can activate themselves by implementing an observation pattern for their communication between each other and with the blackboard. We address this idea in more detail in Section 13.3.

Knowledge sources represent the active components of a blackboard pattern [52]. Since we extend the capabilities of knowledge sources as described in the following, we encapsulate the new functionality in a new subclass called `Kit`. At the same time, we continue using the collective term *knowledge sources* as an abstraction when referring to more than one instance, such as *EmotionKit* and *ThinkingAloudKit*.

Following their activation, kits follow the general blackboard flow: Using the `execCondition` method, they start to `inspect` the blackboard to retrieve currently available solutions. At the same time, the kits start to `monitor` kit-specific usage data. This is a first difference to traditional knowledge sources, as **Cuu** kits are not only able to inspect and update solutions, but also collect data, which results in having multiple raw input streams, rather than only one. A second major difference is the continuous application of `execAction`, once the kit has been activated. This is required since the constant stream of new usage data that results from the monitoring may enable the detection of new solutions. We summarize the action execution which reflects the main competence of a kit in Figure 7.2.

As every kit provides its own raw usage data stream, they apply their expertise through processing usage data (represented by multiple incomplete symbols in Figure 7.2) into usage information (represented each by a single, complete symbol in Figure 7.2), which they then relate to the feature under development in order to derive usage knowledge (represented by a complete symbol which is surrounded by a frame, i. e., the feature, in Figure 7.2). The involved methods `process` and `relateTo` are summarized within `extract`. The action execution always results in calling `updateWorkbench`, which places the latest solutions on the workbench. While doing so, each kit is able to retrieve existing solutions from other kits through the `inspect` method from the workbench. If they find a solution that they can use for improving their own solution, they `fuse` the solution which internally triggers a new call of their process method; i. e., `fuse` is a higher-level version of `process`. We developed four **Cuu** kits and provide an in-depth description of them in Chapter 8—including a more general description of their capabilities.

In comparison to the original pattern description [52], we concede more capabilities to the blackboard class, which is why we provide our own abstraction: the `Workbench`. The workbench stores `solutions`, which can be of different granularities. In Figure 7.2, we adapt the representation used by Lesser *et al.* [190] to depict the different granularities following the level terminology. The lowest level of a solution is the input stream that originates from the monitor methods of the kits, which use both hardware and software sensor for usage data collection. We call usage data a `Level 0` solution. Likewise, we refer to usage information as a `Level 1` solution. Starting from `Level 2`, we refer to solutions as usage knowledge which may be *of a higher level*. The workbench also stores data about the current feature under investigation within `featureData`. To provide access to this data, the methods `configure` and `read` are used. The management of a feature relationship is a major extension of the workbench in comparison to the traditional blackboard.

The solutions provided by the kits might be combined in multiple ways, leaving the developer with different ways of alternatives of how to interpret the user's tacit knowledge in the moment of observation.[9] The developer's goal is to find an optimal solution to represent the user's tacit knowledge provided the investment of a reasonable time, following the description of forces by Buschmann *et al.* [52]. This uncertainty is characteristic for the application of the blackboard pattern: The kits support the creation of an approximation for the tacit knowledge, which can be refined, *verified*, and used by the developer. The developer is able to ac-

[9] Developers interpret the results of knowledge sources to develop a new product increment. This can constitute a non-externalized decision-making process. The ability to reconstruct it in hindsight may be valuable, underlining the importance of *Rationale Management* for user understanding. The extraction of tacit knowledge reflects a *wicked problem* [255] or *ill-structured problem* [215, 222], that requires collaborative discussion for solving: the knowledge sources act as such participating discussion entities.

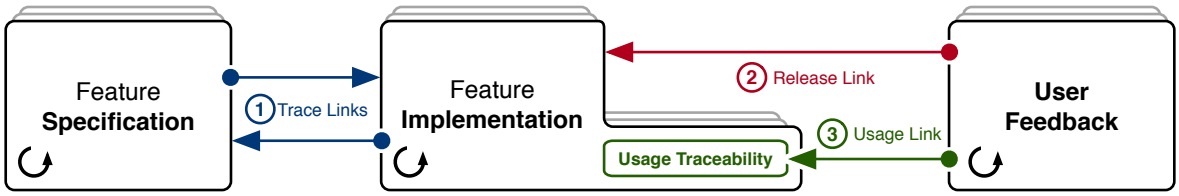


Figure 7.3: Different link types of artifacts, adapted from Johanssen *et al.* [155].

cess the solutions through a dashboard, which is an extension of the blackboard that provides a visual representation of the usage knowledge. We detail this aspect in Section 7.3.2. In choosing a combination of solutions from different knowledge sources—in particular when a kit can no longer deliver a higher-level solution provide the currently available solution—the developers themselves become a *control knowledge source* [52]. This knowledge is manifested in a well-defined combination of knowledge views in the dashboard, as described in Figure 9.6. We rely on the proposed usage of the `controlData` attribute of the blackboard for storing this knowledge. In the future, this control data may be shared between developers for reuse.

The conceptual use of a blackboard pattern for extracting tacit knowledge from usage data helps us to further identify limitations and considerations regarding different aspects, which we describe along the *forces* listed by Buschmann *et al.* [52]: First, given the *immaturity* and vagueness of the domain, the combination of solutions from kits is done on an *experimental* basis; there is potentially no definite solution that applies across multiple different features. With each new release, a new assessment of the provided solutions needs to be performed by the developer. Second, the solution of a kit represents an answer to a *partial problem*. Therefore, solutions from multiple kits should be combined to better understand users.

7.2 Usage Monitoring

The frequent release of software increments fosters the use of implicit user feedback which is collected through the application of usage monitoring. The utilization of monitored usage data, in particular with respect to experimentation as described in Section 2.1.4.2, forms a broad research field [260]. The main benefit of implicit feedback lies in the ability to collect usage data continuously without interference with users; the results support developers in the decision-making process on how to improve a feature under development [196]. However, a challenge remains in relating processed usage information to the feature increment's particulars; we use Figure 7.3 to elaborate the challenge.

*Feature Specification*s, such as requirements in form of user stories or scenarios, are codified in a *Feature Implementation* that can be released to users. A relation between these two artifact types can be established through *Trace Links* ①; one example for the implementation of trace links are feature tags [285]. A second example is the use of feature branches, i. e., their names or the issue identifier that lead to the creation of the feature branch, which points to the feature implementation [167].

After the *Feature Implementation* is released to users, *User Feedback* can be collected. For explicit usage data or information, such as a written user feedback on an app store, it may be possible to understand and derive the user's reference point from the content of the feedback. Guzman *et al.* show that a feature relationship can be derived in textual feedback [133]; for instance, Example 6 in Section 2.1.3 depicts an example of user feedback that encompasses a feature reference. For monitored usage data, this is often not possible. As a result, the usage data can only be collected on a coarse-grained representation, such as the entire implementation as shown in Figure 7.3 with the *Release Link* ②. This is similar to the second example for trace links, in which the implementation's feature branch serves as an identifier, the release link. This is, however, not enough to create a relationship between the user feedback and the implementation's particulars.

To address the lack of reference points, we introduce *Feature Crumbs*. Feature crumbs form a usage monitoring concept that allows the allocation of usage data to a feature particular. We adapt the terminology from Gotel & Finkelstein [122] of requirements traceability, when we describe feature crumbs as a means to create a *Usage Traceability*. In Figure 7.3, we illustrate usage traceability as the result of a *Usage Link* to maps usage data to code in a backward direction as shown with ③.

In practice, feature crumbs reflect software sensors that are manually seeded into the application source code: They should be placed at a location which reflects a relevant characteristic, i. e., the *particular* of the feature under development. Thereby, they describe a user-centric structure of the feature, since they are called during the run-time of the application execution. Feature crumbs enable a fine-grained assessment of feature usage; they form the basis to derive usage information, such as whether a user (a) started, (b) canceled, or (c) successfully finished the usage of a feature. Additional usage information from kits, as introduced in Section 7.1, can be recorded in parallel to the observation of feature crumbs to enrich the assessment of a feature. For example, user-related characteristics such as their emotions, derived from *EmotionKit* (Section 8.2), can be aligned with detected feature crumbs to precisely reveal situations in which users were confused.

At the same time, feature crumbs enable an assessment beyond the pure analysis of the user feedback: by combining both the usage- and the trace link, the decision-making process regarding the implementation of a feature can be related to the monitored usage data of this feature and vice versa. This relationship is use-

ful to evaluate and iterate on previously made decisions. In addition, the frequent iterations that are characteristic for CSE are illustrated with the multiple instances and the spinning circle arrow in Figure 7.3: Not only the feature implementation and the feature usage data continuously change, this also holds true for the feature specifications: Based on the monitored usage data, the discovery of additional requirements or the refinement of existing requirements becomes possible. Feature crumbs for usage monitoring in CSE affect the whole software development process, which we describe—besides requirements and details regarding feature crumbs–in the following.

7.2.1 Requirements

There already exists work that transfers common software engineering practices to the frequent software engineering domain. For example, Kleebaum *et al.* defined requirements that need to be met to integrate decision knowledge into CSE [168]. Following their approach, we elicit six requirements that need to be fulfilled in order to adapt usage monitoring in a CSE environment. These requirements are formulated from the developer's perspective, who is the main stakeholder; their needs serve as the basis for the feature crumbs, which we introduced in Section 7.2.2.

Feature Assessment

The monitoring concept shall enable developers to determine if a feature that is currently under development and released to the user was actually used by the user. This enables the developer to only consider user feedback that was provided by a user who used a particular feature—and omit other user feedback. This reduces the amount of user feedback the developers need to investigate while they are evolving the feature. In addition, performance indicators that can be derived hereon shall support the developer in understanding the acceptance of the feature. For instance, a performance indicator may include measurements such as how often a feature execution is canceled by users.

Usage Data Allocation

The usage of a feature typically includes the utilization of multiple user interface elements, such as a sequence of buttons that users tap on to achieve a goal. A monitoring concept shall allow the allocation of observed user feedback to a specific step of a feature. This enables a detailed feature analysis. For example, the developers can study the time frames between feature steps to speculate about the users' impression at that point in time. In addition, the relationship enables the transformation of usage information to usage knowledge, and thereby allows to combine the user feedback with other knowledge sources.

Feature Flexibility & Comparability

CSE is characterized by frequent and rapid releases of new software increments which typically introduce only minor changes. This requires the monitoring concept to be a simple extension to the feature under development and to enable a flexible adaption to new functional additions. At the same time, the flexibility of the concept shall contribute to the consideration that user feedback from consecutive feature increments need to be comparable. The developer shall be able to investigate the users' feedback with respect to the feature's evolution.

Application Range

The applicability of a monitoring concept shall be independent of the software type it is used in: As CSE is often used for the development of mobile interactive systems, the concept shall be at least applicable to graphical user interfaces. At the same time, it shall also be possible to monitor computationally intensive code that does not directly interact with users. For future use cases, the concept shall be applicable to emerging user interfaces, such as voice or augmented reality, as well as be compatible with server-side operations.

Environment Compatibility

Today's development environments can be described as a setting that consists of management approaches, software processes, as well as a variety of tools and platforms. The monitoring concept shall be compatible with existing environments and not impose an additional burden upon developers, for example in terms of tool adaption. The monitoring concept shall be applicable to any management framework that is applied by a development team.

Effort & Learnability

The monitoring concept shall be easy for developers to learn and apply. This is in particular important as developers refrain from using complex and heavy-weight additions to their development process. The concept shall impose a minimal cognitive load upon the developer who is responsible for adding and maintaining the means for usage monitoring. This ensures seamless utilization of the concept.

7.2.2 Feature Crumbs

Based on the elicited requirements, we present an object model of our monitoring concept in order to introduce feature crumbs and its related classes. Furthermore, by describing the implications of feature crumbs for the development process in, we highlight its lightweight character.

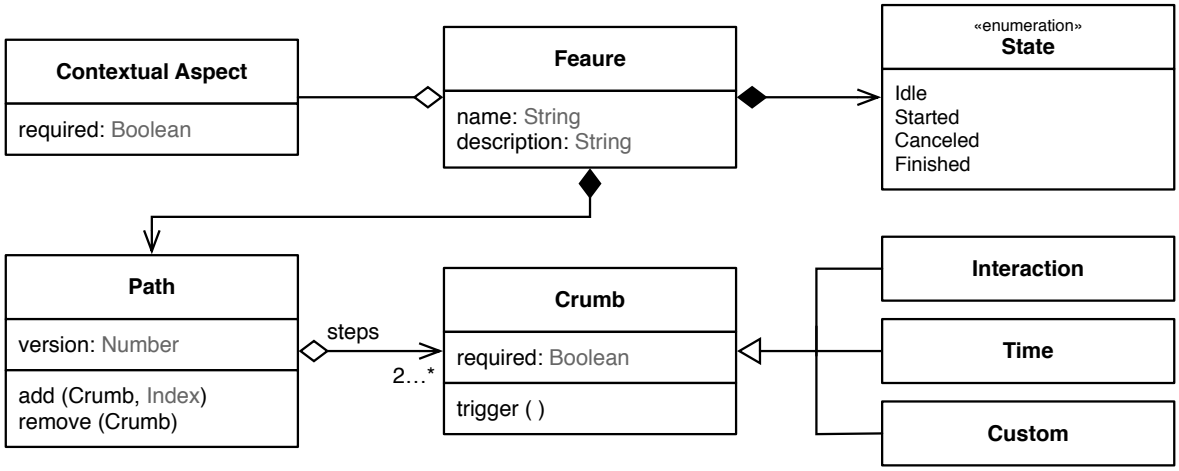


Figure 7.4: The object model that describes the feature crumb concept, adapted from Johanssen *et al.* [155] (UML class diagram).

Object Model

We depict the major entities of the object model in Figure 7.4. A `Feature` consists of a `name` and a `description`. For every feature, there can be one or more feature `Path`s. A path is always exclusively defined for one feature. It consists of a sequence of `steps`, which represents a chronologically sorted array of at least `Crumb`s which specify a flow that needs to be completed by users in order to execute a feature. As the implementation of a feature is refined, i. e., extended or updated, crumbs can be `add`ed or `remove`d from a path to reflect the code refinements. In case the feature refinements are major improvements, the add and removal methods lead to an increment of the path `version`.

In case a feature is composed of optional steps, the respective crumb's attribute `required` might be set to *false*. This allows to construct more complex feature paths, as some features may include optional aspects that are not part of every feature execution. We distinguish between different classes of crumbs. All of them share the capability to `trigger` an event, however, they differ in the condition that activates the trigger. An `Interaction` crumb triggers when the user interacts with the interface, which can be any interaction or event that occurs during the run-time of an application. A `Time` crumb encompasses logic to defer its trigger call until a specified time frame has passed. This crumb class is useful to simulate situations in which the users consume content, such as when reading a large text segment on the screen. `Custom` crumbs conform to any developer-designed conditions. For instance, the input of a string into a text field that matches a pre-defined sequence option may trigger an event. The `State` of a feature relates to its observed execution. Based on the recorded events that are triggered by crumbs, their occurrence in the feature path can be verified. Based on wether or not they are part of the feature path, we can then distinguish the following feature states: (a) *Idle*, i. e., the feature

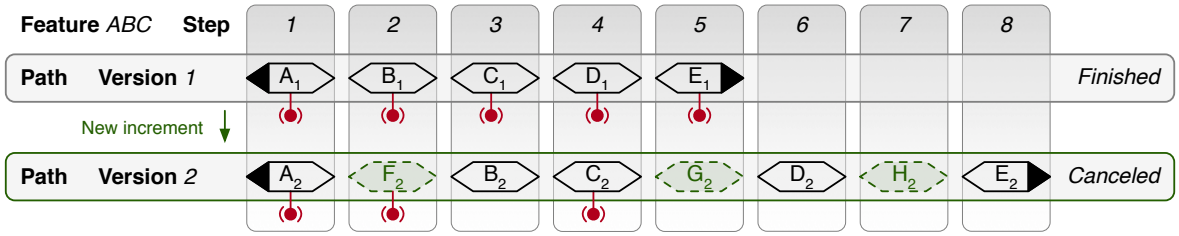


Figure 7.5: Informal representation of the feature crumb object model that is shown in Figure 7.4, adpoted from Johanssen *et al.* [155].

has never been initiated by the user; (b) *Started*, i. e., the first crumb in the sequence of the feature path of a feature was detected; (c) *Canceled*, i. e., the assessment of a feature path is stopped due to a stop criterion. This might be a defined time frame or a special application event, such as the application was closed; or (d) *Finished*, i. e., all crumbs that are part of the feature path sequence were detected sequentially. Eventually, we modeled `Contextual Aspects` that describe conditions that need to be met in order to start the feature path observation. These may be pre- or post-conditions, such as an external event or the availability of certain user data. Since contextual aspects are optional, we consider them out of scope for this work.

Generic Object Model Instantiation We sketch an informal representation of the feature crumb concept in Figure 7.5 to illustrate the objects that are described in Figure 7.4. The `Feature` entitled *ABC* was improved once. As a result of the new increment, `Path` was updated with a new `version` number. Version *1* consists of five feature `Crumbs`, which are depicted as a hexagon. Black corners on the left or right side indicate the first or last crumb of a `steps` sequence, respectively. Version *2* introduces three new crumbs that are highlighted with a dashed, green border. They occupy steps that were previously allocated to other crumbs. Both versions were released to users for testing. The red antenna signals that during runtime of the application, a crumb was `triggered`. For version 1, as all crumbs were triggered, we conclude that the feature has been executed properly, i. e., it is *Finished*. Version 2 was *Canceled*, since a stop criterion was found: C_2 was triggered before B_2.

Specific Object Model Instantiation We can describe the versions depicted in Figure 1.1 using the same informal notation. For an approximation of where to allocate the feature crumbs, we use the description in Example 3 as well as the speculation of how users might have reacted. In Figure 7.6, we sketch the flow of feature crumbs for each feature path version. For all versions, crumb A represents the event when the map view appears; this simulates the start of the feature, as its goal is to provide a simple way of altering the map layer, which is embedded in the map view. In version 2, B represents the second crumb which is triggered when the user taps

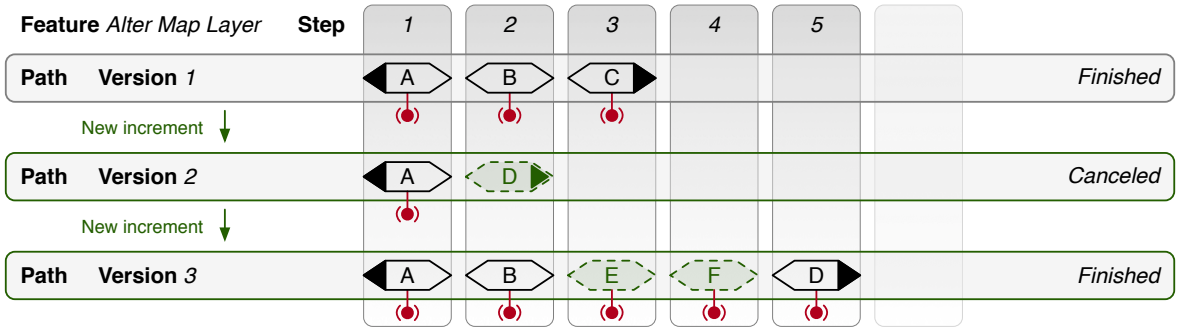


Figure 7.6: Informal representation of feature crumbs that may be used to described Example 3 using the same notation as in Figure 7.5.

on the hamburger menu; it is part of the call that also animates the slide-in effect of the side menu. Crumb C finally represents the event that is triggered with the selection of one of the layer options. From the red antennas we see that all crumbs were successfully triggered and the developer can assume that a user was able to finish, i. e., complete the feature. The new increment that contains version 2 of the feature introduces the new round-circled button for layer selection within the map view and removes the *old* menu entries. To reflect these changes in the feature path, the developer removes the crumbs from the path and adds a new crumb D to both the feature path and the source code—there, it is part of the action call that follows the button tap event. From the (missing) red antenna, the developer sees that, while the map view is presented to the user, they never trigger the newly introduced button. The feature state is marked as *canceled*, as the user never completes the feature in its current version. In version *3*, the developer decides to reintroduce parts of the old implementation; they add a new crumb E for the new menu entry that shows the tooltip once tapped, which triggers crumb F on appearance. Crumb B and D can be reused from version *1* and *2*, respectively. From the state observation, the developer derives that now, the user is successfully able to complete the feature.

Implications for the Software Development Process

The introduction of feature crumbs and the adaption of usage monitoring to CSE has an impact on the overall software development process. As outlined in Table 7.1, one reason can be found in the fact that usage monitoring follows multiple phases and ends one iteration of a development cycle by verifying the product increment. We use the table to list the major capabilities required by developers, resulting artifacts of each phase, as well as the tools and platforms that support the developer in artifact creation while cycling through the software development process. The individual capabilities, artifacts, and tools are interweaved and may be based on their predecessor. In Table 7.1, we highlight entries that are related in one way or the other to usage monitoring, and elaborate on them in the following.

Table 7.1: Outline of the major capabilities, artifacts, and tools across the phases of a software development process, which build on the categorizations of Bosch [41] and Bruegge & Dutoit [45]. The table is adapted from Johanssen *et al.* [155].

Phases	**Requirements Elicitation**	**Implementation**	**Testing**	**Deploying**	**Usage Monitoring**
Capabilities	Elicit & implement requirements, design run-time feature		Design and assess test cases, analyze crash reports		Evaluate feedback
Artifacts	User story, Scenario	Branches & Commits	Test Cases	Crash Reports	Feature Crumbs
Tools	Project Management System	Version Control System	Integration System	Delivery and Distribution System	User Understanding System

A software development process is typically started and dependent on a phase of requirements elicitation for a feature (Section 2.1.4.1. The management is bundled in a project, which can be tracked in project management systems, such as issue and knowledge tracking systems. In such a tool, a feature is described for example as a use case or as a scenario. Given its lightweight character, feature crumbs are compatible with different concepts of how a feature is designed, implemented, managed, or tracked during design-time. Notably, as there is a similar structure of subsequent events, feature crumbs promote the use of scenarios. Scenarios represent a concrete instance of a use case [45] and are therefore very similar to an actual feature observation; this makes them an ideal template for the application of feature crumbs. This supports their usefulness for collecting and analyzing usage data [217].

The choice of an adequate branching strategy can positively affect the work with feature crumbs: We propose to rely on a branching model that uses feature branches for implementation work [176]. This allows the integration of knowledge into CSE [153], which we envisioned in Section 3.2. Feature crumbs may only be added to feature branches to maintain feature atomicity. This also avoids interference with the development of other features. We propose to remove the feature crumbs from the code basis as soon as the branch, on which the development is performed, is merged back to the master, i. e., main, product branch. By following this rule, we highlight the intended use for feature crumbs, which shall only support short-term development which is focused on feature particulars. This stands in contrast to measure and act on long-term effects; which might not even be possible if the feature crumbs remain in the code basis, as the related code parts may change over time, and no longer reflect the initial feature that was described with the feature path.

During requirements elicitation and feature implementation, the developers must design a run-time representation of the feature which is currently under development. Only in doing so, they can make use of feature crumbs to enable evaluation

of the feature during usage monitoring. This requires an additional effort for developers during both requirements elicitation and implementation. We use the following analogy to support this aspect: Adding feature crumbs is similar to writing software test cases—for which developers have to reflect on the feature before coding [68]—to be able to verify the implementation afterwards. For example, a test assertion can be as trivial as `assert(true==false)`. While this assertion reflects a syntactically correct test case, its added value is in doubt. Similarly, the insertion of feature crumbs needs to be well-chosen by developers to be of actual use.

Only after a software increment is tested and delivered to a target environment, a usage monitoring phase that involves feature crumbs can be performed. To instrument, i. e., analyze the monitored usage data, the developers are in need of tool support in the form of a user understanding system. This includes multiple aspects, such as the collection, management, and assessment of feature crumbs, but also the management of other usage knowledge from other sources. This can be understood best by referring to the explanation of the methods shown in Figure 7.1. The `release` method captures the moment when the developers finish their development on the software increment, enriched the increment with feature crumbs, and defined the feature path—all of which is completed by the release of the increment via continuous delivery to the user. The configuration of the user understanding system reflects the declaration of the feature so that the usage knowledge captured by other kits can be aligned to and compared with each other. The release method is called every time a new feature path version is created, which requires a reconfiguration of the feature stored in the workbench. We present an implementation of the user understanding system in form of the **Cuu** workbench in the following Section 7.3. This enables developers to evaluate monitored data and reflect upon the product quality.

7.2.3 Related Concepts

There is many research in the area of requirements engineering that addresses the understanding of feature structures. For example, Bakiu and Guzman describe how to automatically detect feature relationships [21]. Similarly, Guzman and Maalej presented a natural language classifier to extract features from reviews [133]. Rooijen *et al.* provide more details on user descriptions and behavior and link it to a requirement [304]. The presented approaches are able to automatically derive features, while feature crumbs rely the manual interaction of developers. In contrast, however, feature crumbs provide insights into actual usage behavior of users.

Generally, the feature crumb concept forms a lightweight task model to describe user interactions. There is a dedicated research domain that addresses the modeling of user tasks. For instance, feature crumbs share similarities with the *ConcurTaskTrees* specification [241], which introduces four task types: a *user*, an *abstract*, an *interac-*

tion, and an *application* task. Given their manifestation in code, feature crumbs do not distinguish *who*, i. e., the system or user, triggered the crumb. This is the reason why, in Figure 7.5, we apply the same symbol—the hexagon, which is used for application tasks by Paternò *et al.* [241]—for all classes of feature crumbs, as described in Figure 7.4. Another difference can be found in the fact that feature crumbs focus on applications developed during fast-cycled processes, such as CSE. Moreover, feature crumbs rely on a linear path, rather than hierarchies or logical relationships.

Johnson *et al.* describe the use of a task model in the context of rapid prototyping [161]. They place their work in the context of scenarios [161], which reflects a similarity to feature crumbs, as they can be derived based on a scenario as well. In contrast, however, their approach is driven by the goals of users, and depict their resulting task knowledge structure in a tree-hierarchy [161], why the feature paths are built as a linear structure.

To implement actual run-time observation, current development environments[10] rely on code additions that are similar to feature crumbs in order to enable developers to create success and cancel paths. These concepts are, however, designed for debugging and profiling applications, rather than focusing on the user experience as it is the main goal of feature crumbs. On the other hand, platforms[11] that apply similar concepts for usage monitoring only address single events and do not promote the addition of other knowledge types.

Finally, various approaches exist that utilize usage data for software evolution. For instance, the UI-Tracer [136] supports developers in comprehending a software system by automatically identifying source code that is related to user interface elements. Similarly, feature crumbs can be understood as the addition of traces to the source code to relate user elements with their implementation. In contrast, however, our concept is not limited to user interface elements. Furthermore, feature crumbs describe features with the goal to collect usage data, which, in turn, can be linked to other knowledge types, such as decision knowledge.

7.3 Workbench

The fast and rapid iterations on software increments [41] during CSE is enabled by activities such as continuous integration and continuous delivery [109]. Each software increment carries knowledge—in particular regarding the user acceptance, which can be derived based on the user feedback, but also regarding its implementation and decision that were made during development. In order to work on and improve the next iteration of the software increment, developers rely on the availability of such knowledge. This, however, represents a challenge: the frequency,

[10] https://developer.apple.com/videos/play/wwdc2018-405/?time=1097
[11] https://docs.microsoft.com/en-us/appcenter/analytics/event-metrics

extent, and complexity of knowledge that is produced during the CSE process increases and is further amplified by a variety of different knowledge sources.

We already highlighted the need of integrating usage and decision knowledge into CSE [153], refined our ideas with the help of practitioners [157, 158] and detailed the results, i. e., how developers should be able to access, visualize, and analyze knowledge, in Section 3.2 and Section 6.2. We focus on knowledge for which CSE enables new opportunities with respect to its collection and integration. For instance, usage knowledge benefits from the rapid release of new increments. Enabled through continuous delivery, users are always guaranteed to provide feedback on the latest version of the software; it enables the instant replacement of outdated software. As for another example, decision knowledge benefits from repeating practices, such as merging branches or changing the status of issues which reflects the project progress. Decisions are then documented in commit messages or in issue comments, respectively. To enable access to data that result from CSE activities, such as continuous delivery, or tools and platforms, such as version control systems, an additional component is required that collects and manages such data. In addition, such a component needs to be able to provide access to the results of knowledge sources and prepare them for visualization.

In this section, we introduce the **Cuu** workbench, which consists of a platform for the integration and management with CSE activities, as well as a dashboard as the core component for knowledge visualization. The dashboard enables developers to access any type of knowledge that arises during CSE. We focus on the work with usage knowledge and rely on decision knowledge to exemplify the extensibility of the workbench. Furthermore, as we discuss in our previous work [153], we pose that the joint consideration of usage and decision knowledge benefits the development process by increasing the software quality. A combined visualization in a dashboard can act as a means to fuse both knowledge types.

7.3.1 Requirements

We list requirements that shall be met by the **Cuu** workbench. The requirements are fulfilled either by the platform (Section 7.3.3) or the dashboard (Section 7.3.2).

Knowledge Visualization

The workbench shall create an environment which enables access to knowledge that is currently inaccessible to developers. This includes knowledge that was previously stored at locations separated from each other. There shall be a functionality that offers the addition or removal of different perspectives on a knowledge type at hand. This is done under the assumption that the workbench offers the ability to select a feature under development for inspection by the developer.

Knowledge Interaction

We suggest that the workbench should offer the possibilities for visually interacting with knowledge types. This may also include basic commands that are frequently performed by developers' during their daily work, which could be enriched by the ability to add or refine knowledge.

Knowledge Investigation

The workbench shall offer the ability to select the scope for which knowledge is presented; there might be different abstractions which support this selection process. For instance, a commit might serve as a specific relation, i. e., a point in time, to determine the scope for which knowledge should be presented. This should be reflected in a main component of the workbench, that is part of the main interaction with the developer. Each component shall represent an individual perspective on a software project. The selection of the scope shall trigger associated views to update their knowledge visualization.

Knowledge Comparison

The workbench shall provide the ability to select two reference points of a software project for which knowledge should be compared. This enables developers to reflect on whether a particular implementation change led to change toward the desired user feedback, for example in a change in their behavior. The comparison shall be supported by an adequate visualization.

Knowledge Evolution

The workbench shall provide the ability to select an interval of software project states. This enables developers to derive trends in knowledge evolution, such as an increased acceptance by the users. The evolution shall be supported by an adequate visualization.

Feature Crumb Management

To enable usage monitoring that follows the feature crumbs concept introduced in Section 7.2, the workbench shall be able to create a feature path for a feature under development. This requires the management of a feature representation within the workbench, i. e., its creation, naming, and deletion. It shall be possible to configure the feature path by adding, relocating, and removing feature crumbs. The configuration shall be simple and intuitive. The workbench shall store feature path and related feature crumbs in an adequate format. It shall further be clearly visible

which software increment contains which feature version, and whether this version has been used by users or not.

Project Management

The workbench shall be able to manage access to different projects, which act as the container for multiple features that are part of an application under development. The project shall be independent of the target environment, i. e., whether the application is developed in a mobile or desktop context.

Platform Integration

The workbench shall enable the integration of knowledge management into CSE activates; this requires its ability to connect and interact with other platforms and tools that are applied in the context of CSE. Support for relating code repositories to a project in the **Cuu** workbench is indispensable and shall be easily to perform for developers. As they reflect the main reference point for code changes, information on new software increments shall be transmitted and connected automatically to a project in the **Cuu** workbench. The interaction with the code repository and the **Cuu** workbench shall be lightweight and flexibel to use for developers. The process shall adopt to the flow of the development process of the project and developers may decide whether to investigate on knowledge of a feature or not. Other platforms for integration with the **Cuu** workbench might be issue tracking systems, which allow the collection of decision knowledge. In general, the workbench shall be extensible through an easy to use plug-in system to exchange data with current platforms and future ones that might become relevant.

User Feedback Processing

A core competence of the **Cuu** workbench is to provide an interface to receive usage data, usage information, and usage knowledge from knowledge sources. On the one hand, this is reflected in the ability to receive and directly store user feedback from knowledge sources that process the user feedback directly on the client device on which the user feedback is collected or monitored. On the other hand, this may require to provide an environment for extended user feedback processing by a knowledge source; for example if usage data over time needs to be collected before usage information can be derived. For these cases, the **Cuu** workbench shall offer the ability for knowledge sources to perform calculations within the platform. In order to process the user information, either on-device or within the platform, the **Cuu** workbench shall offer an interface to provide information about the current feature path and feature crumbs.

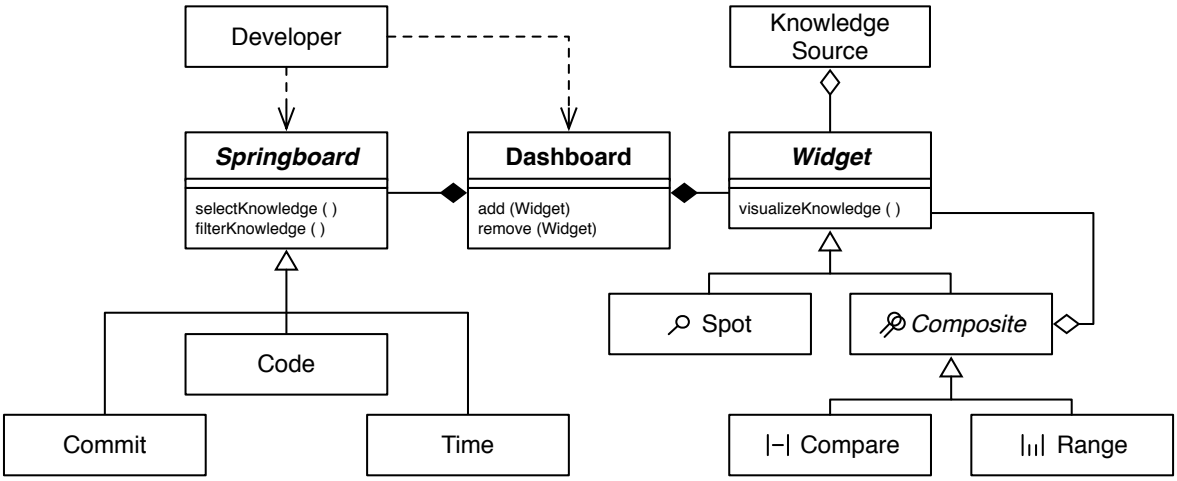


Figure 7.7: The main abstractions of a dashboard, namely Springboards and Widgets (UML class diagram). We introduce icons for the widget classes which are used in the following sketches. Adapted from Johanssen *et al.* [154] (© 2017 IEEE).

Role-Independent Usage

As we learned from the interview study in Chapter 6, the use of the **Cuu** workbench, i. e., the dashboard, shall not be limited to solely developers. While we remain with the developer as the main stakeholder of the dashboard, every team member shall be able to access the dashboard and inspect knowledge. Therefore, there shall be no burden that restricts individuals from accessing a project.

While a simple user management is required to maintain project separation, adding a new member to a project in **Cuu** shall be simple and easy to follow. This shall put the focus of the **Cuu** workbench on *what* can be done with the knowledge, not by *whom* it is done.

7.3.2 Dashboard

We present general concepts for a dashboard that can be used to visualize knowledge that is produced and collected during CSE. The four major goals of the dashboard are to allow developers to *follow*, *reflect*, *interact*, and *react* on knowledge available in CSE environments. In the following, we first provide an object model of the dashboard and then detail its entities by sketching its visual components.

Object Model

Our concept of a dashboard for knowledge visualization consists of two major abstractions: `Springboard`s and `Widget`s. In Figure 7.7, we depict a class diagram with the conceptual structure of both abstractions. Furthermore, the relationship to the main stakeholder, the `Developer`, is shown.

Springboard enable developers to specify and thereby control the scope of knowledge that will be visualized in widgets. As the developer must be able to specify the knowledge of interest, at least one springboard instance must be present in the dashboard at any time. Each springboard reflects a software development project from a different perspective, i. e., a view of interest. In Figure 7.7, we propose three springboards that we consider the most basic classes: `Commit`, `Code`, and `Time`. Each springboard class manifests itself in a specific relation to a knowledge type and thereby qualifies as a knowledge selector. For example, a commit springboard is beneficial for inspecting usage knowledge, while a code springboard might be more adequate to inspect decision knowledge; we provide more examples in Section 7.3.2.3. Using the inheritance relationship of the springboard as shown indicated Figure 7.7, new springboard classes might be added if they offer the ability to `selectKnowledge`, i. e., the core responsibility of a springboard, as well as to `filterKnowledge`, i. e., defining constraints that facilitate the knowledge selection. Both methods are initiated by the `Developer` who is using the dashboard.

Developers can also `add` and `remove` one ore more widgets to the dashboard, depending on their knowledge needs. A widget represents a knowledge-specific view that is offered by a knowledge source; we provide specific examples in Section 7.3.2.4. Following the selection of knowledge through the springboard, widgets `visualizeKnowledge` related to the selection. We differentiate between three `Widget` classes:

- **`Spot`**: This class allows access to knowledge that is characterized by its relationship to a point in time, a version of an artifact, or an event such as a commit—generally any knowledge that can be related to a particular reference point. This may also include knowledge that needs to be aggregated over a period of time to provide additional insights, as long as the aggregated result can be related to a reference point.

- **`Compare`**: This class enables developers to contrast knowledge from two different dates or events; in other words, comparing the results of two individual knowledge states that could be presented in a spot widget. This allows developers to explore difference in the performance of two alternative implementations. Thereby, when comparing usage knowledge, this widget class enables the discovery of an optimal solution when choosing between two implementation alternatives.

- **`Range`**: This class supports developers in inspecting the evolution of knowledge. This the range typically relates either to a period of time or to the aggregating of knowledge across a collection of events.

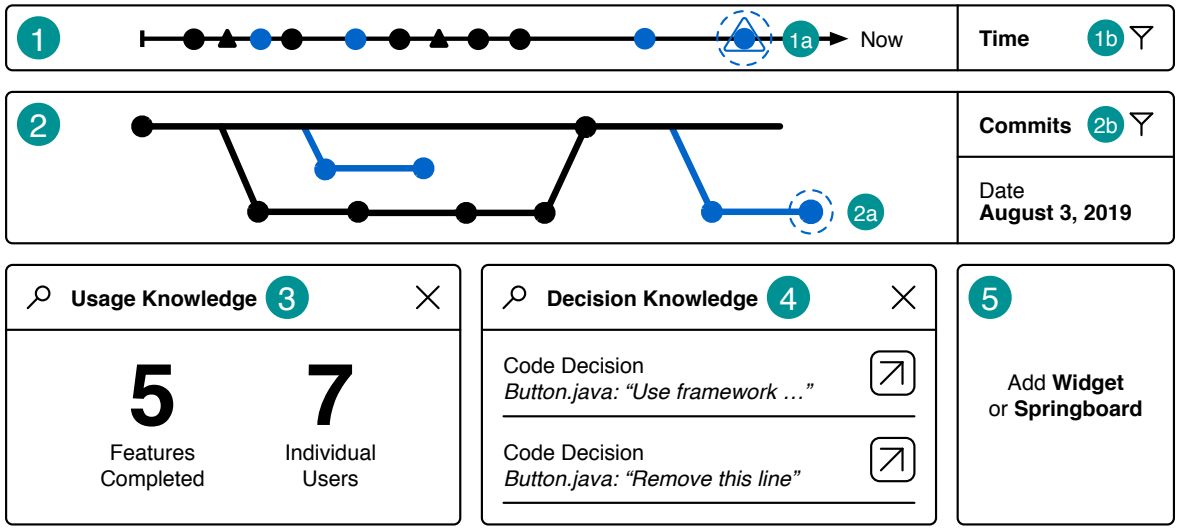


Figure 7.8: Dashboard example, adapted from Johanssen *et al.* [154] (© 2017 IEEE).

As indicated in Figure 7.7, the implementation of compare and range widgets follows a composite pattern that relies on the spot widget as a leaf class. This enables the creation of different manifestations of complex widgets in future scenarios. For instance, a widget that enables the comparison of three or more alternatives at the same time can be created. A visionary scenario might even incorporate potential, not yet existing knowledge within a widget. For usage knowledge, this may be a widget that compares actual usage knowledge with usage knowledge that would result from a potential change—in case there is a respective knowledge source that can predict such usage knowledge. The widget would be reflected in a subclass of the compare class, allowing the assessment of the effect of changes without applying them.

Widgets and their content are provided by the knowledge source. Every instance of a widget conforms to a defined interface to visualize its domain-specific knowledge, such as usage knowledge or decision knowledge. Additional subclasses enable widgets to amplify this knowledge. We detail the structure and instances of knowledge sources in Chapter 8.

Dashboard Example

We propose a grid-based layout for the dashboard which we illustrate in Figure 7.8. The sketch shows a dashboard hosting a time springboard (Figure 7.8-①) and commit springboard (Figure 7.8-②) Both springboards indicate that an entry for knowledge visualization has been selected; this is depicted by a dashed circle (1a and 2a). Additional filters are activated individually for a dashboard (1b and 2b). Two spot widgets, one for usage knowledge ③ and one for decision knowledge ④, present their content in relation to the selection.

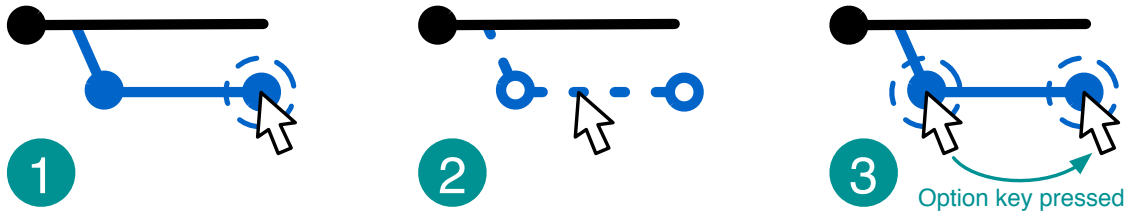


Figure 7.9: How to trigger a spot, range, and compare widget within a commit springboard, adapted from Johanssen *et al.* [154] (© 2017 IEEE).

Additional springboards and widgets can be added by developers from a list of available widgets and springboard: They are found behind a dedicated interface element (Figure 7.8-⑤). Both springboards occupy fixed positions in the dashboard and cannot be relocated; only one of them can be removed, as there needs to be one springboard present at any time. Widgets, however, can be moved and resized along the dashboard grid. This provides developers with the flexibility to adjust the dashboard to their needs, creating an environment for exploring knowledge.

The consideration of multiple knowledge widgets supports the developer during the development process, toward the goal of increasing the software quality. For instance, a low number of feature completions can indicate that users are not able to use the feature; the reason might be found in a decision that was made within the latest release. At the same time, a positive user feedback strengthens the benefits of the latest improvements and serve as a pro-argument for a decision that was made beforehand. Overall, the joint visualization facilitates the discovery and utilization of such relations. Therefore, we further explore more examples of individual widgets in the following sections.

Springboard Examples

Springboards are the main interaction element in the dashboard. They specify the knowledge that widgets prepare for visualization. On a technical level, their effect is similar to the `grep` command in Linux. Springboards can be differentiated by their perspective on a software project. Following Figure 7.7, we propose three springboards classes, namely a commit, code, and time springboard—additional springboards are possible.

In the commit springboard in Figure 7.8-②, black lines and circles indicate existing development branches and commits, respectively. Open branches and related commits are represented by blue color. The three widget classes can be activated using the interaction patterns described in Figure 7.9.

If the developer taps on a commit, this commit is selected as a reference point. This is indicated by a dashed circle around the commit (Figure 7.9-①). This action notifies all spot widgets that are added to the dashboard, provides them with the

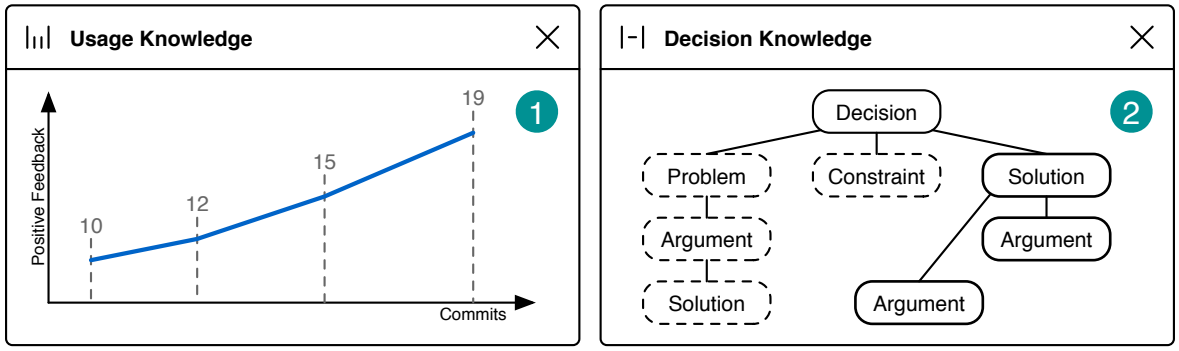


Figure 7.10: Examples for a usage knowledge range widget and a decision knowledge compare widget, adapted from Johanssen *et al.* [154] (© 2017 IEEE).

commit information as a reference point, and asks them to update their content according to the selection. Multiple commits can be selected by pressing an additional key, such as the option key, and then tapping on the commits of interest (Figure 7.9-③). This initiates the visualization process for all compare widgets. The activation of a feature branch, which is performed by a tap on the line that connects two commits, selects the feature branch as a period of time or a collection of events (Figure 7.9-②). This initiates the visualization process for all range widgets.

The time springboard sketched in Figure 7.8-① works similar to the commit springboard. Individual commits are depicted as circles and distributed on a horizontal line, forming a timeline. In contrast to the commit springboard, the time springboard uses the commits' timestamps as the main reference point for presentation. Hence, this springboard is able to select knowledge that is not introduced by or bound to a specific commit by adding additional entities on the timeline. For instance, black triangles represent decisions that were made at a specific point in time, preceding or following a commit. In case decisions are documented in a commit message or as a code comment, a triangle is added around the related commit circle as shown in Figure 7.8-(1a).

Finally, by using springboard filters, developers gain the ability to further detail the selection scope. For instance, the filter of a time springboard in Figure 7.8-(1b) can limit the timeline to only show reference points of the last week. Similarly, a springboard filter in Figure 7.8-(2b) could be used to show only commits that are concerned with a particular decision.

Widget Examples

In this section, we further exemplify how knowledge types are represented in spot, compare, and range widgets, using Figure 7.8 for spot widgets, as well as using Figure 7.10 to provide examples for a range and compare widget.

Usage Knowledge Usage knowledge widgets visualize user feedback that follows the usage of a software increment. We introduce four knowledge sources for user feedback in Chapter 8; each of them provides a domain-specific widget that can be added to a dashboard.

In addition, we envision to integrate existing approaches into widgets to visualize explicit user feedback: As introduced by Guzman *et al.*, a spot usage knowledge widget may include a visual representation of rating or sentiment distribution [130]. Our dashboard extends their approach by controlling the reference point of the visualized feedback using springboards.

Visualizing implicit user feedback is of great interest for CSE. As shown in Figure 7.8-③, a spot usage knowledge widget provides insight into the development of a feature on a commit, such as usage knowledge in form of the amount of features that were completed or individual users that interacted with the feature. This knowledge reveals whether the feature is adapted by users or not. Based on such insights and in accordance with decision knowledge that is presented in another widget, developers can reconsider their implementation decisions.

A range usage knowledge widget as shown in Figure 7.10-① provides insight on user feedback over time. If a feature branch is selected, its containing commits are displayed on the x-axis, while for example the amount of positive user feedback are shown on the y-axis. This supports the developer in assessing whether the improvements appeals to the users.

Decision Knowledge Decision knowledge widgets' competence is to visualize decisions that were made throughout the software lifecycle. This relates to artifacts such as requirements or code. The decision knowledge needs to be retrieved from multiple sources, such as issue trackers, code repositories, or wiki platforms. The collection is performed by the platform, as outlined in Section 7.3.3.

CSE enables developers to continuously reflect on decisions [167]. Decision knowledge spot widgets allows them to revisit such decisions, as sketched in Figure 7.8-④.

Developers may also rely on a decision documentation model that encompasses different components, such as the problem that needs to be solved or multiple proposals for a solution [141], As these models are incrementally refined throughout the development process, they are suitable for being presented in a compare widget to contrast decision components at two points in time. We sketch an example visualization in Figure 7.10-②: Dashed border lines indicate earlier documented decision components, whereas the solid border indicates those documented recently.

To understand changes of decisions over time, developers rely on range widgets. Existing approaches may be integrated for that purpose: For example, a range widget could visualize the evolution following the suggestion of a spiral by Zhi and Ruhe, in which new decision components are visualized along its curves [323].

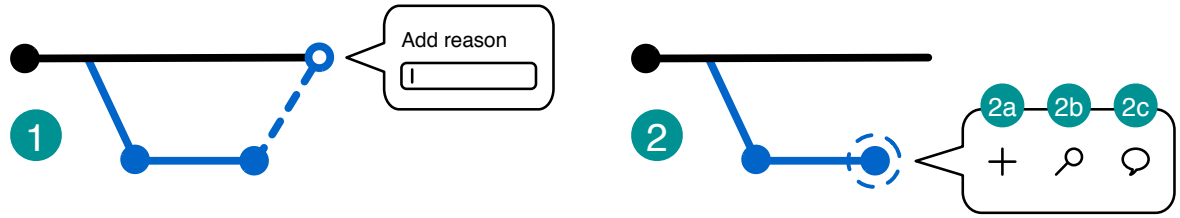


Figure 7.11: Two examples for interacting with the commit springboard, adapted from Johanssen *et al.* [154] (© 2017 IEEE).

Interaction

The visualization of existing knowledge is the core functionality of the dashboard. In addition to that, it provides the possibility to visually support the creation of new, the maintenance of existing, and management of relationship between any type of knowledge. We propose that springboards act as the main point for such interactions. In Figure 7.11, we sketch two examples for this interaction within the commit springboard.

As sketched in Figure 7.11-①, a *pull request*, i. e. the process of merging back code from a feature branch into the master branch, can be achieved by dragging a commit to a branch. In the following action, a popup can be used to specify the reasons why the code in question is merged into another branch. In general, we do not intend to replace existing tools for these interactions. Instead, by providing such an additional way to trigger CSE-related activities [167], the dashboard allows developers to add and enrich knowledge in a convenient way.

In Figure 7.11-②, we sketch more actions that can be triggered from a commit: First, the attachment of additional knowledge to the commit (Figure 7.11-2a). Second, the instantiation of new widget for a detailed analysis of the commit (Figure 7.11-2b). Third, the proactive request of explicit user feedback for the commit (Figure 7.11-2c). This, reflects an interaction point with the end users as they are invited to use a new release from a commit. At the same time, a new usage knowledge spot widget could be added to the dashboard that keeps track of the feedback that results from that action.

Discussion

We discuss the dashboard and its visualization approach on the basis of our four goals, i. e., to enable developers to *follow*, *reflect*, *interact*, and *react* on knowledge. We provide a visual overview in Figure 7.12.

CSE encompasses activities that lead to constant change. This is for example reflected in branching strategies. Practitioners attest difficulties for some developers to keep track of the changes and the overall mentality of the project, as reported in Observation 10. As the dashboard allows developers to visually follow code

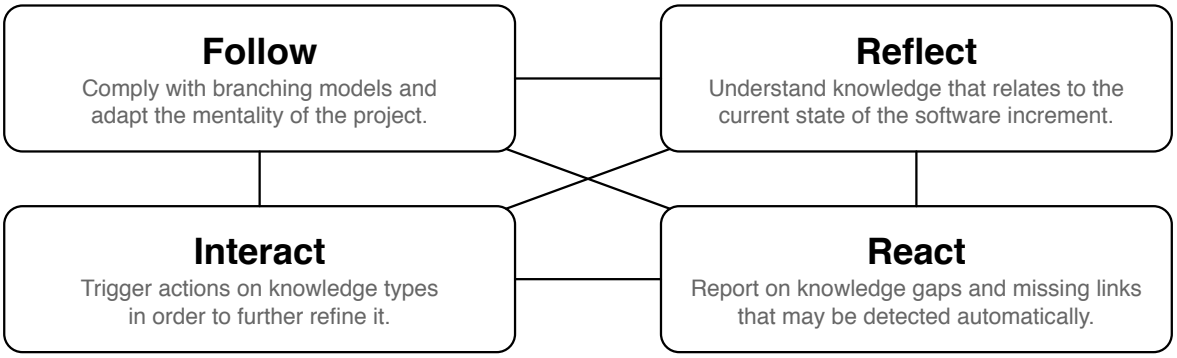


Figure 7.12: Four goals of the dashboard that relate to each other.

changes, especially the ones they have not been involved in, it goes beyond sole knowledge visualization and supports developers in their daily work.

An important goal of the dashboard is to enable developers to reflect on knowledge related to the feature under development. By using springboards and widgets the developers are able to reflect not only about an individual knowledge type, but also about relationships between knowledge. This enables them to understand the reasons for the current state of the software increment. Notably, a widget's expressiveness, i. e., its usefulness for the reflection process, directly depends on the quality of the widget's visualization. Therefore, metrics that capture the knowledge content of a widget should be investigated to determine the expressiveness of widgets. For instance, the number of visual elements that are required to express a widget's content could be considered for this purpose. At the same time, the result of the reflection process depends on the developers' capability of discovering correlations between the knowledge visualized in the widgets; we further discuss this process in Chapter 9. To support inexperienced developers, it is helpful to allow experienced developers to link widgets that have proven to be valuable in joint usage. Such combination of widgets is reflected in the *controlData* of Figure 7.1.

The dashboard encourages developers to interact with existing knowledge. We outlined first ideas to allow an intuitive and convenient way of interaction toward the connection and addition of new knowledge using springboards. Still, other concepts to enrich existing knowledge within the dashboard should be investigated.

This interaction enables the goal of allowing developers to react to reports on knowledge gaps and missing links. The visualizations that we presented so far rely on structured knowledge, in real CSE environments, however, it might not be well-structured: for instance, branch relations might become complex and difficult to understand [93]. To increase the usefulness of the dashboard, these gaps should be detected automatically. Thereby, developers are supported in understanding knowledge and detecting missing knowledge elements, creating a hybrid approach to resolve problems. This goal addresses the Observation 44 of practitioners' reports.

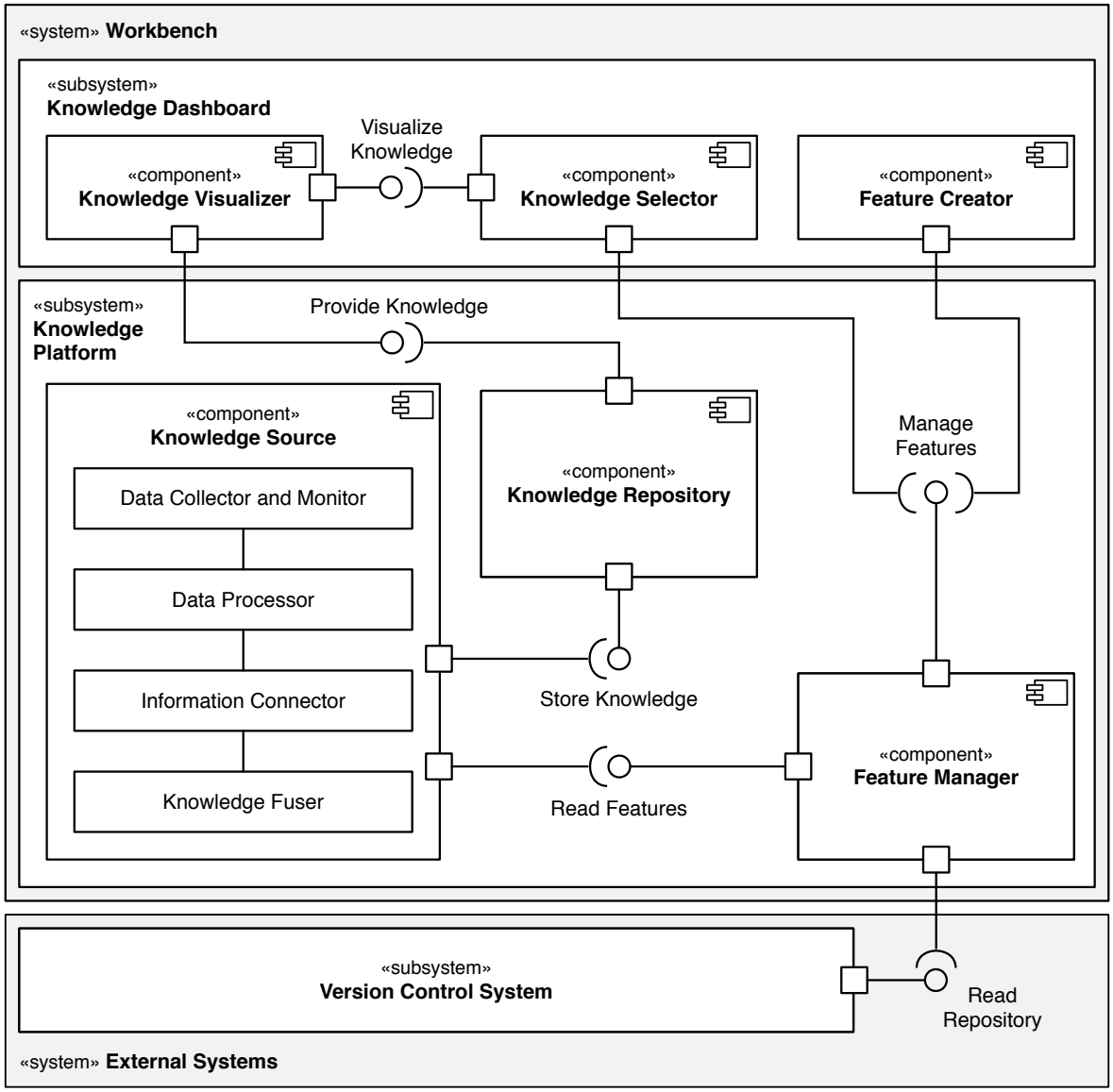


Figure 7.13: The services of the **Cuu** workbench (UML component diagram).

7.3.3 Platform

The **Cuu** workbench bundles two subsystems, i. e., the dashboard and the platform, to enable access to usage knowledge. It is designed for deployment in an CSE environment, which we illustrate in Figure 7.13.

The `Knowledge Dashboard` subsystem presents knowledge to the developer; we detailed the inner composition of this subsystem in the previous Section 7.3.2. Widgets are represented by `Knowledge Visualizer` components, while it is the `Knowledge Selector` component that encapsulates the functionality of a springboard. Thereby, this subsystem reflects the visual representation of a blackboard.

The technical representation of the backboard can be found in the `Knowledge Platform` subsystem. This subsystem is essential as it fuels the **Cuu** workbench by

providing access to persisted knowledge and facilitating the communication with external systems. The `Knowledge Repository` acts as a traditional blackboard on which knowledge can be stored by `Knowledge Source` components. Knowledge sources encompass logic to collect, process, and relate user feedback, as well as fuse different knowledge types. The components' competence may be instantiated on different physical devices: while the `Data Collector and Monitor` are typically part of the application and therefore run on the same device that executes the application, other logic of the component may be performed remotely. Knowledge sources can be of different types, that are not necessarily associated with usage knowledge. As described, they might for instance also provide insights on decision-making procedures. In this case, they are also considered an external system.

The `Feature Manager` component is in charge of handling **Cuu** features, i. e., the management of feature paths and their crumbs in relationship to a software increment. Therefore, this component uses services of the external subsystem `Version Control System`. By reading the repository, the feature manager is able to understand the latest commits separated by branches. The feature manager offers two services to **Cuu**-internal components: On the one hand, knowledge sources can read the features, i. e., path and crumb information. This enables them to transform usage information into usage knowledge. On the other hand, the feature manager offers services to the knowledge selector to enable the selection of which knowledge should be presented, as well as to a `Feature Creator` component, which represents a visual interface for developers to define feature path and crumbs.

7.3.4 Foundations

To the best of our knowledge, the visualization of knowledge in CSE has not yet been explored in previous research. This holds true in particular for the visualization of different knowledge types such as usage and decision knowledge at the same time.

We found work regarding the individual visualization knowledge types. With respect to usage knowledge Guzman *et al.* [130] provide an approach: They propose visualizations to make user feedback accessible. In addition, analytic platforms provide detailed insight into the application usage. In general, knowledge on users' interaction forms the basis to reiterate user interface designs [189]. With respect to decision knowledge, Lee and Kruchten [187] as well as Shahin *et al.* [287] addressed approaches. Hesse *et al.* [141] used the DecDoc tool to visualize single design decisions as a tree of decision components such as arguments or alternatives. Notably, this work focuses on design decisions and not on decisions regarding software artifacts that were developed during CSE.

Sharing and understanding knowledge in agile software development is important, yet difficult to implement [239]. Alperowitz *et al.* worked on a set of metrics to make knowledge in agile project courses measurable and provided a visualiza-

tion in the form of a report sheet [10]. Elsen introduced a tool to visualize complex branch structures in git environments [93]. The presented work shares a focus on the visualization of specific aspects. However, it differs from the work presented in this dissertation: We introduced a dashboard that allows for a comprehensive visualization of different knowledge types, such as usage and decision, that relate to software artifacts during CSE.

With respect to the platform, we want to highlight work by two researchers: Röhm introduced the MALTASE framework to follow user interactions and help developers utilize the resulting information [259]. Pagano proposed the PORTNEUF framework to encourage user feedback and provided a method to assess the importance of individual feedback [235]. Similar to the work presented by Röhm and Pagano, we focus on a platform to support developers.

7.3.5 Implementation

We implemented the workbench[12] to as a web application, which allows access through a web browser. This addresses multiple requirements up front. Most of the platforms in the setting of CSE rely on web interfaces to offer services to their users, the developers. Following the requirements elicited in Section 7.3.1, the choice of a web application for the **Cuu** workbench not only guarantees the ability that the developers can include it within their workflow, but also the burden for technical communication is reduced, as we can make use of similar technologies. Also, with respect to support multiple stakeholders, a web platform increases the accessibility for other stakeholders of a project since a web application can be used platform independent and its usage follows common practices. In the following, whenever it contributes to the understanding of the artifact, we continue the example of Section 1.1 to exemplify the development of the map layer selection.

Project Overview

Figure 7.14 shows a screenshot of the project overview screen which is presented to the user after they authenticated themselves.

It contains a list of all **Cuu** projects that the developer can interact with; every row reflects a project as the highest-level entity to bundle feature paths, feature crumbs, and their related knowledge. Every **Cuu** project relates to a code repository project, which hosts the code for the application and respective features. In the project overview, a **Cuu** project[13] provides developers with access to the three main

[12] In the following, we add the prefix **Cuu** to workbench whenever we want to highlight that we refer to the artifact's implementation of the **Cuu**$^{\textsf{SE}}$ framework.

[13] In the following, if not stated otherwise, we omit the prefix **Cuu** when referring to a project to facilitate readability. The prefix is only used to distinguish it from projects that are managed in other systems, i. e., when the terminology is overloaded.

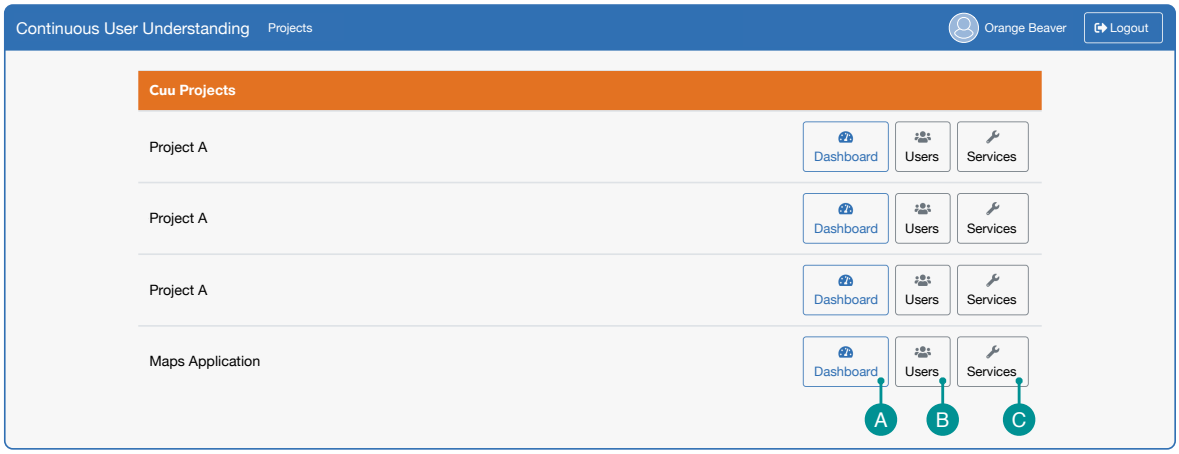


Figure 7.14: Screenshot of the project overview screen in the **Cuu** workbench.

activities of the workbench, which we detail with screenshots in the following: interacting with the *Dashboard*, managing permissions for *User*, and configuring *Services*.

Dashboard

Figure 7.15 shows a screenshot of the main dashboard screen, which developer rely on to visualize and interact with knowledge. The dashboard is accessed per project, as indicated in Figure 7.14-Ⓐ.

The dashboard hosts many interactive elements to visualize and interact with knowledge. In the top left corner, the name of the project is presented (Ⓐ). We adapt the concept that we describe in Section 7.3 by dividing the dashboard into the two main entities, springboard (the top part of the screenshot, i. e., Ⓒ) and widgets (the lower part of the screenshot, i. e., Ⓚ).

We implemented a commit springboard, as it offers the most flexibel and detailed way of selecting the scope to be presented for usage knowledge visualization. The developer may use a dropdown menu to select the feature they want to inspect (Ⓓ). In Figure 7.15, the *Map Layer Selection* feature is selected. We depict that the related code is developed on a feature branch called *feature/MAPS-21-layer-selection*, right below the dropdown menu. The feature branch name matches the actual branch name in the code repository to make it easy for developers to identify their working branch when they switch between the **Cuu** workbench and their code or development environment; as a result, this name serves as a reference point and cannot be changed. The name in the dropdown menu reflects an **Cuu**-internal name which can be changed by the developer, using a dedicated button Ⓕ. The fact that a different name can be chosen allows developer to emphasize that they are focusing on a particular aspect, maybe even within the feature itself, that they are trying to investigate on. In our case, a more specific name might be *Extract Layer Selection to Circle Button*. A new **Cuu** feature is added by button Ⓔ and detailed in Figure 7.16.

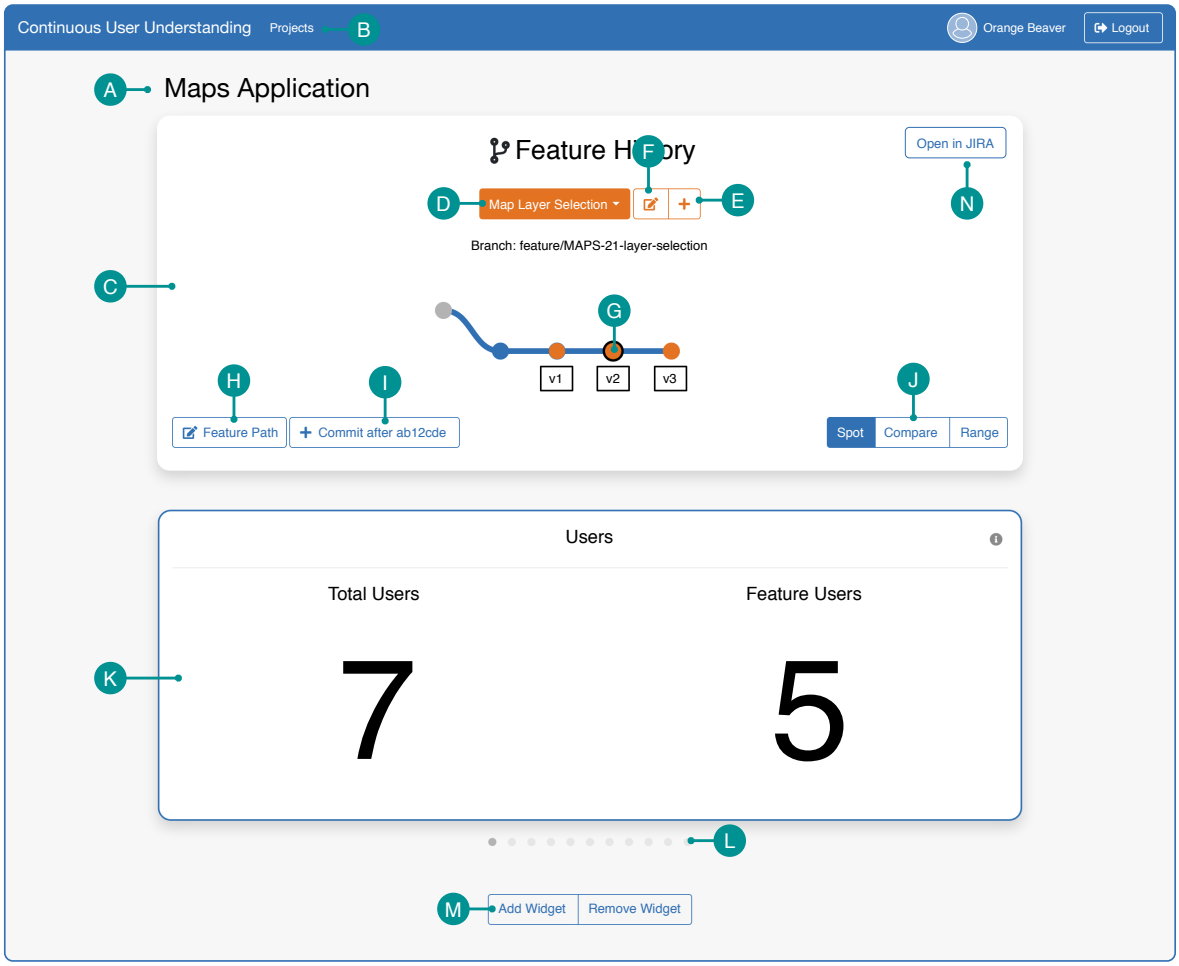


Figure 7.15: Screenshot of the dashboard screen in the **Cuu** workbench.

The center of the springboard hosts a branch representation of the code commits that match the evolution of the feature in the code repository Ⓖ. A blue circle reflects a commit, an orange circle a commit that has been released and used by users. Following the concept in Section 7.3, a selected commit is outlined with a black boarder. Hovering over a commit tiggers a tooltip that allows the developer to retrieve more context, i. e., better relate the commit to its code counterpart. The badge below the commits indicates the feature path version, as introduced in Figure 7.4. The example in Figure 7.15 imitates a development progress that might have followed for Figure 1.1. Ⓗ enables developers to define a feature path; we detail this process in the following screenshots.

Using the segmented controls in Ⓙ, the developer controls the classes of widgets that are displayed, i. e., spot, compare, or range. The widget area in Ⓚ updates according to the defined class; in case the developer selects compare or range they need to pick two commits in Ⓖ.

The widgets area in Ⓚ is organized as a horizontal carousel, in which the developer latches individual widgets either by swiping to the left or the right with the

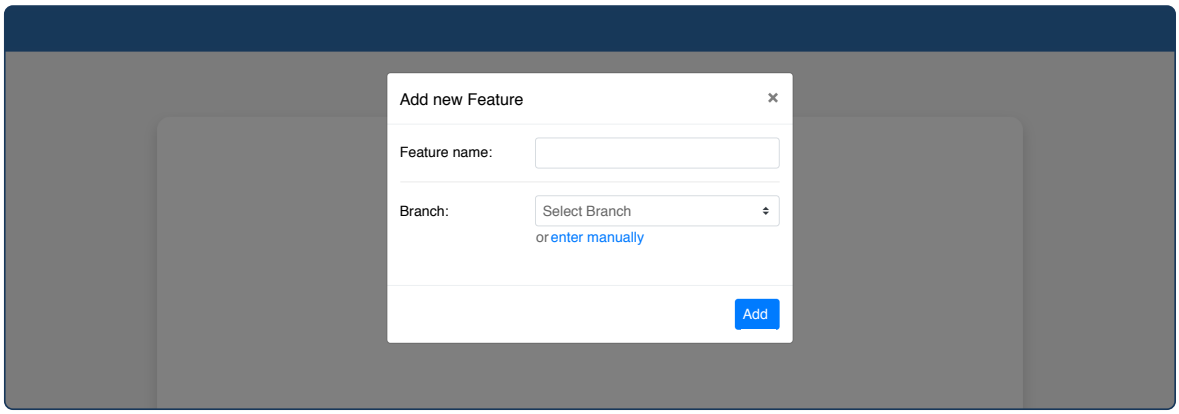


Figure 7.16: Screenshot of the dashboard overlayer which can be used to add a new feature in the **Cuu** workbench.

mouse or with their fingers on a touch device, by using the left or right arrow key, or by selecting a particular position that is represented by a dot below the widget (Ⓛ, this dot area also indicates the current index of the widget displayed). Developers stack multiple widgets by using the add and remove buttons in Ⓜ. A new row of widgets consists of the identical set of widgets, however, allow the developers to *mix and match* multiple widgets and inspect their content by skimming the website vertically. We discuss this method in more detail in Section 9.2.2.

The addition of a new **Cuu** feature to the dashboard is performed manually by the developer, since not every development branch needs to be represented within the **Cuu** workbench. In Figure 7.15, Ⓗ enables developers to activate a feature for usage knowledge investigation. Figure 7.16 shows the overlay that is triggered by the button; as described, the developer enters a name which best suites the purpose for investigation and can then select a code branch which should be associated with the feature. The branch name is selected from a list of code branches that the **Cuu** platform is familiar with; we discuss their source of origin in the last description of screenshots when discussing *Services*.

The link to a branch may also be added manually. This allows a more flexible way of defining the code reference, which can be used in case the automatic detection of branches was incomplete, or in case the developer want to setup a particular branch as a new **Cuu** feature. A similar purpose is achieved with Figure 7.15-Ⓘ, which allows to manually insert a commit after the selected commit using its hash value. Overall, the ability to manually modify the knowledge reference in form of commits addresses the requirements as described in Section 7.3.1 and enables more flexible and dynamic process flows.

In Figure 7.15, the developer can select a commit (Ⓖ) for which they want to define a feature path (Ⓗ). Figure 7.17 shows a screenshot of the overlayer that will be triggered. It shows the name of the feature for which a feature path will be defined. The expected input is a sequence of text strings that the developer adds via

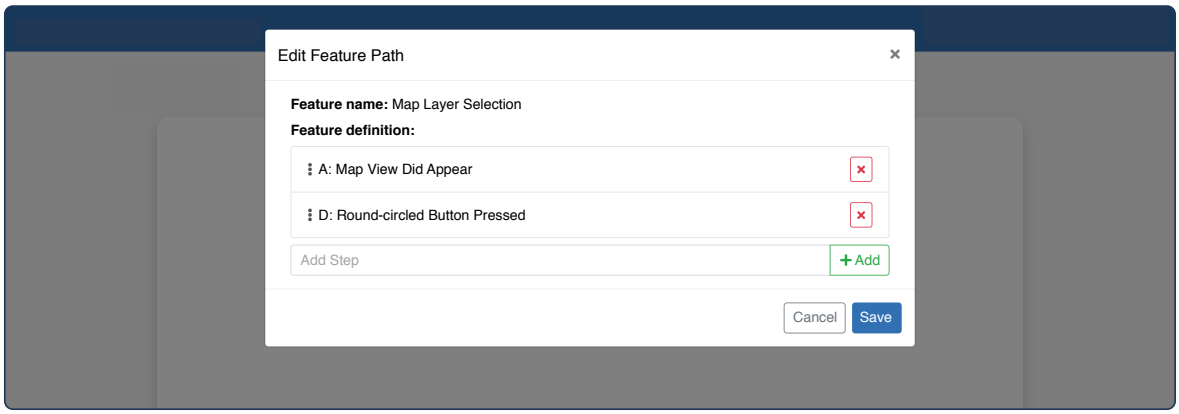


Figure 7.17: Screenshot of the dashboard overlayer which can be used to add and modify a feature path in the **Cuu** workbench.

the text field—the default text prints *Add Step*. The text needs to match the name of a crumb that is seeded within the commit for which the feature path is defined. New crumb names are added via *+Add*, existing crumbs can be deleted via *x*. The order of crumbs can be modified via drag and drop. The feature path is stored after a click on *Save*. Following the specific example in Figure 7.6, the feature path of *version 2* consists of two feature crumbs which may be entitled *A: Map View Did Appear* and *D: Rounded-circle Button Pressed*.

Users

Figure 7.18 shows a screenshot of the user management screen, which developers rely on to oversee and manage users that have access to the project. The screen is invoked per project, as indicated in Figure 7.14-Ⓑ.

The **Cuu** workbench is developed on a prototypical basis, intended for research purpose. Therefore, we can rely on a minimal user management for the **Cuu** workbench. We use the Shibboleth[14] system as it is a wide-spread approach for a single sign-on solution in the context of an academic infrastructure. We configure Shibboleth in a way that we retrieve a minimum amount of information that ensure the anonymous use of the platform by developers. This puts the focus of the platform of *how* knowledge is accessed, not by *whom*. Only an identifier that is unique per user and an instance of the **Cuu** workbench is exchanged after a successful authentication. This has the following benefits: First, user do not need to create new credentials to be able to use **Cuu**. Second, since the user name and password are verified in an external system that is not part of the **Cuu** workbench, as well as no person-related information is stored, the risk of security issues is reduced to a minimum. Since there is no user name received from Shibboleth, we rely on automatically generated user names, such as *Orange Beaver* or *Blue Dog*.

[14]More information on Shibboleth is available at https://shibboleth.net.

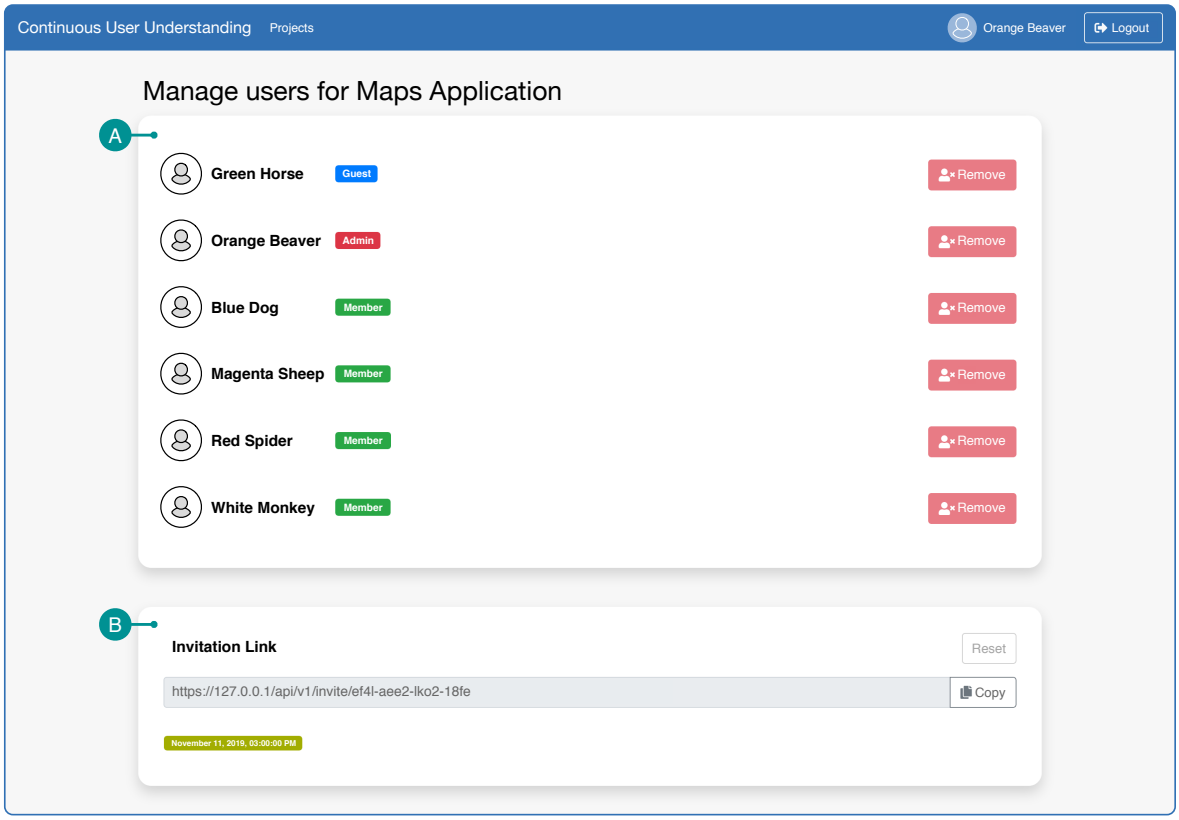


Figure 7.18: Screenshot of the user screen in the **Cuu** workbench.

As shown in Figure 7.18-Ⓐ, a list of users per project indicates who has access to the dashboard. User can be removed from a project using the *Remove* button. The **Cuu** workbench offers a basic role hierarchy, differentiating between *Admin*, *Member*, and *Guest*. While their use is not a requirement of the workbench, it does enable providing access for the sole purpose of consuming knowledge, rather than setting up services (as described in the next session) or managing the user permissions.

New users are added to a **Cuu** project using a link-based invitation system, as indicated in Figure 7.18-Ⓑ. The link is unique per project and does not change over time, however, it can be reset. The link can be send to fellow team members or other stakeholders via email or an instant messaging service.

Services

Figure 7.19 shows a screenshot of the service management screen, which developer rely on to oversee and manage connections to other **Cuu** components and external systems. The screen is invoked per project, as indicated in Figure 7.14-Ⓒ.

In Figure 7.19, the connections to other, external systems can be configured. This reflects components indicated as external systems in Section 7.3.3. In order to functioning correctly, two services need to be registered with the **Cuu** workbench: the **Cuu** SDK as presented in Section 8.1.2 and a version control system.

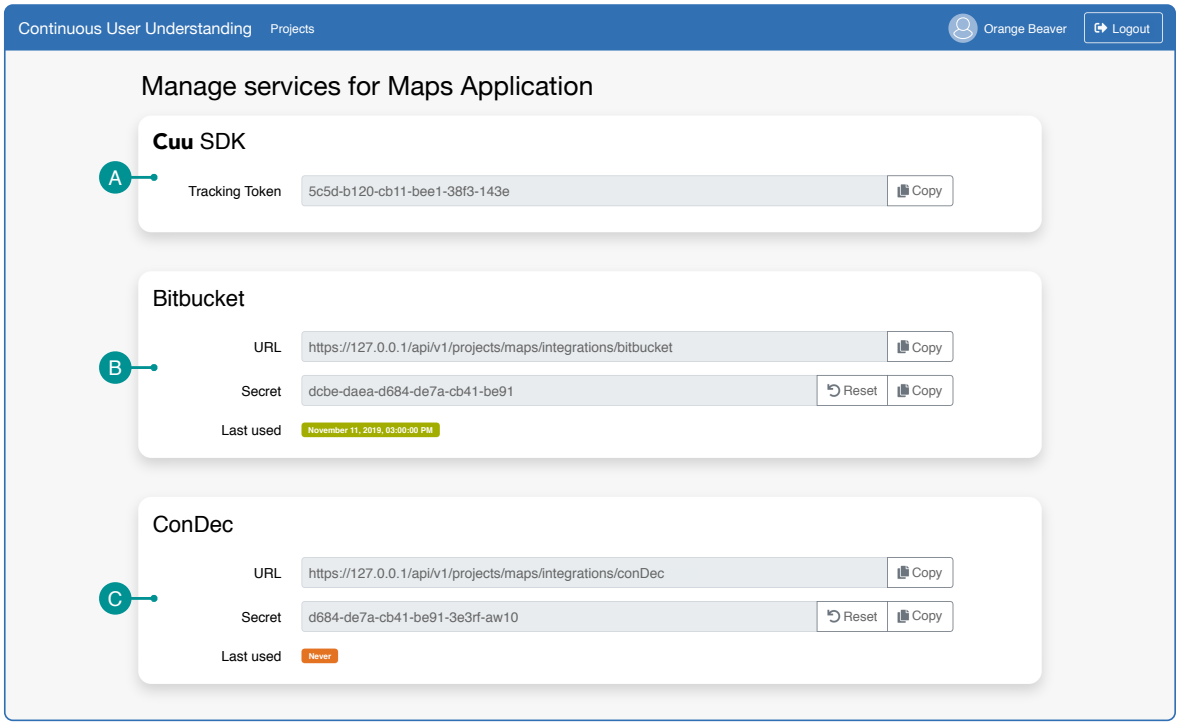


Figure 7.19: Screenshot of the services screen in the **Cuu** workbench.

In Figure 7.19-Ⓐ, a *Tracking Token* is displayed. In combination with the name of the **Cuu** project, this tracking token needs to be added by the developer to the configuration of the client-side **Cuu** SDK. The SDK can then send user feedback that is monitored and collected by kits to the respective **Cuu** project (Section 8.1.2).

Likewise, in Figure 7.19-Ⓑ, we display configuration information that can be used to connect *Bitbucket*[15], a platform to manage and share git repositories. The Uniform Resource Locator (URL) and the secret enables Bitbucket to continuously push information about new commits and feature branches to the **Cuu** workbench

The connection to Bitbucket implements a webhook system, which follows the result of the interview study with practitioners and is detailed in Example 9. In a version of **Cuu**, that was also used during the instrumentation validation described in Chapter 11, developer were required to add references to commits manually. Following the initial feedback as described in Chapter 11, we added the webhook system. However, this bears some challenges. On the one hand, as developers can create multiple local commits and push the results to a remote. On the other hand, git offers a variety of operations on a repository, such as deleting commits as well as merging or rebasing branches. Eventually, the webhook may be activated during the project, which leaves some commits unknown to **Cuu**. As a result, we faced challenges in supporting every edge case in the prototypical implementation of the **Cuu** workbench. Therefore, while we support the automatic addition of commits

[15]Bitbucket is developed by Atlassian, more information is available at https://bitbucket.org.

and feature crumbs, we continue to offer the manual addition of as describe above. Furthermore, as presented in Figure 7.15-Ⓚ, we only show a single branch in the springboard and not a full representation of the repository as proposed in Figure 7.8.

If available, the services screen allows to connect to other external systems that follow the same webhook system as described for code repositories. For example, with Ⓒ, we show how *ConDec* can be connected and serve as an additional knowledge sources that provides decision knowledge that can be presented a widgets inside the dashboard.

Chapter 8

Knowledge Sources for Continuous User Understanding

> *"Have you ever seen one of the people who will be users of your current project? [..] Have you talked to such a user? Have you visited the users' work environment and observed what their tasks are, how they approach these tasks, and what pragmatic circumstances they have to cope with?"*
>
> — JAKOB NIELSEN [216]

This chapter addresses the capture of usage knowledge: The **Cuu**^{SE} framework relies on **Cuu** kits as knowledge sources. **Cuu** kits encapsulate the competence to collect and monitor usage data as well as the expertise to process it toward usage knowledge. In addition, kits may also offer additional functionality, such as on-device presentation of the usage knowledge [129] to increase its understanding, or access to external services, such as instant messaging platforms, which enable the collaborative discussion of usage knowledge [319].

In Section 8.1, we outline the major challenges that knowledge sources in CSE environments have to face. Then, we introduce the **Cuu** Software Development Kit (SDK), which provides the means for developers to make use of the **Cuu** kits within their applications. We also introduce a first set of three kits, i. e., *FeatureKit*, *InteractionKit*, and *NoteKit*; they are the most basic instances of **Cuu** kits and—besides the ability to act as a knowledge source on their own—have the distinction of being a provider of input during the fusion of usage knowledge within other **Cuu** kits.

The remaining sections introduce four additional instances of **Cuu** kits, namely *EmotionKit*, *ThinkingAloudKit*, *BehaviorKit*, and *PersonaKit*. They exemplify the extensibility of the **Cuu**^{SE} framework. The kits are share the requirement of unobtrusive data collection and monitoring, such as harnessing implicit usage data sources which are known from the domain of affective computing or processing usage data automatically by incorporating machine learning techniques.

8.1 Kit Design

This section presents the general **Cuu** kit design. This includes major challenges for knowledge sources. We also detail the **Cuu** SDK, which provides insights on how the kits can be used by developers, and introduce three instances of basic kits.

To describe how developers can make use of the presented **Cuu** kits, we continue the example that we already used as the basis for explanation in Chapter 7. Following the maps example in Section 1.1, when we show a widget, we assume that currently version 2 is deployed to users and the developer selected the respective commit in the springboard as depicted in Figure 7.15-Ⓖ. The release contains two feature crumbs, i. e., *A: Map View Did Appear* and *D: Round-circled Button Pressed*.

8.1.1 Challenges

Knowledge sources face various challenges when monitoring usage data from mobile applications during CSE. They need to be taken into consideration when designing **Cuu** kits, as they impact the usage knowledge extraction process.

Working with a Limited Set of Data

CSE settings are characterized by the fact that only a few users get in contact with new software increments initially. This is why the availability of usage data for the creation of classifiers is limited. The composition of actual users of a software increment might be unbalanced. As suggested in Chapter 5, the first users that produce usage data might be internal sources, such as fellow developers, dedicated test teams, or even the developers themselves. As a result, the monitored usage data might have a bias, which must be considered during usage data inspection.

Dealing with Frequent Changes

CSE promotes the frequent change of the software increments: buttons are added, views are removed, and labels are modified. As a consequence, there is no consistent state that can be referred to as the ground truth. Knowledge sources need to acknowledge this aspect when visualizing the usage knowledge.

Choosing the Appropriate Data Source

Mobile applications provide multiple sources of usage data, such as clicks, gestures, but also input from hardware sensors, such as accelerometer or gyroscope, or from software sensors, such as view events. Consumer devices promises access to a variety of biometric sensors, such as the users' heart rate. The choice of the data source is an important aspect for creating usage knowledge to reflect the users' intentions.

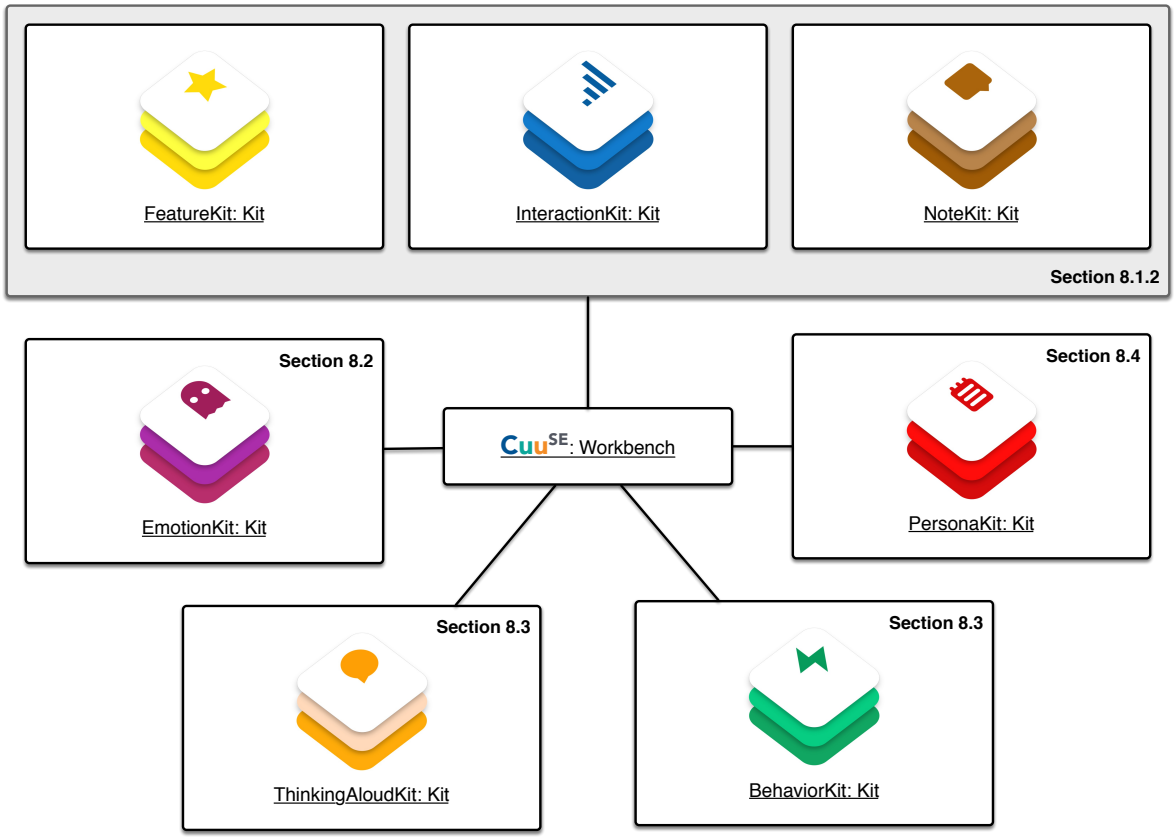


Figure 8.1: **Cuu** kits that are introduced in this chapter (UML object diagram).

Labeling Potentially Noisy Data

While trying to solve a specific task, users produce noisy usage data: for example, they are making use of different features that might be irrelevant for the completion of a task at hand. Also, functionality might be interweaved or distributed throughout the application. This impedes a correct and automatic labeling by knowledge sources. Finally, the granularity of an application's functionality might vary. For instance, a functionality might encompass one or multiple steps to perform. This can have an effect on the precision of results produced by the knowledge source.

8.1.2 Software Development Kit

The **Cuu** SDK enables developers to benefit from the **Cuu** kits. Using the SDK, they gain access to the kits for an application. Figure 8.1 summarizes all seven kits that we developed as part of the **Cuu** SDK and that can be employed by users.

Following the subsystem decomposition in Figure 7.13, every kit should provide either a data collector or monitor, a data processor, an information connector, and a knowledge fuser. The strict distinction between the classes should be continued in the components' implementation. This enforces privacy-related aspects: If the data

processor and information connector are executed locally on the device, they can remove personal data and produce usage knowledge which is then transmitted to the knowledge repository. In general, the **Cuu** SDK is in charge of two configurations, *general* and kit-*specific* aspects, which we outline in the following.

With respect to general configurations, the SDK enables developers to establish the client-side link to a **Cuu** project which can be inspected in the **Cuu** workbench. Using a property list in the application project, the SDK requires the developers to add a tracking token and the project name following the information that is displayed in Figure 7.15-Ⓐ. To initiate the **Cuu** SDK, the developers need to add an `import CUU` statement to their application. In addition, the SDK allows, i. e., requires, the developers to manually start and stop the execution of kits using the method calls `CUU.start()` and `CUU.stop()`, respectively. These methods are usually added during application start and termination and provide the developers with an extra level of control over the SDK.

By default, **Cuu** SDK activates four kits: FeatureKit, InteractionKit, NoteKit, and BehaviorKit—all of which we will describe in more detail in the following. The developers can manually deactivate NoteKit and BehaviorKit, or add more kits; however, FeatureKit and InteractionKit are started at any time to ensure basic data collection. Note that users are presented with an information screen, which we depicted in the Appendix C.1. The screen fulfills the purpose of informing users that they are being observed; an information that they need to acknowledge. Thereby, they can manually deselect a kit in case they do not want it to monitor their application usage. This should be considered by the developer, as it may limit the expressiveness of data provided by the SDK.

With respect to kit-specific configurations, the **Cuu** SDK provides the means to operate individual kits. For example, the SDK provides replacements for common user interface components, e. g., a `CUUViewController`, in order to enable InteractionKit to monitor more information. The usage of such components also enables the technical foundations for kits such as EmotionKit (Section 8.2) to correctly operate. In addition, the SDK guides developers in activating project parameters that are required for making use of hardware-components, such as the microphone for ThinkingAloudKit (Section 8.3) or the camera for EmotionKit.

FeatureKit

FeatureKit is a knowledge source that enables the observation of implicit user events in order to derive information on the progress of a feature's runtime observation. Therefore, FeatureKit implements the feature crumb concept as described in Section 7.2. FeatureKit is part of the **Cuu** SDK[16].

[16]FeatureKit was developed as an open source Apple iOS framework. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/FeatureKit.

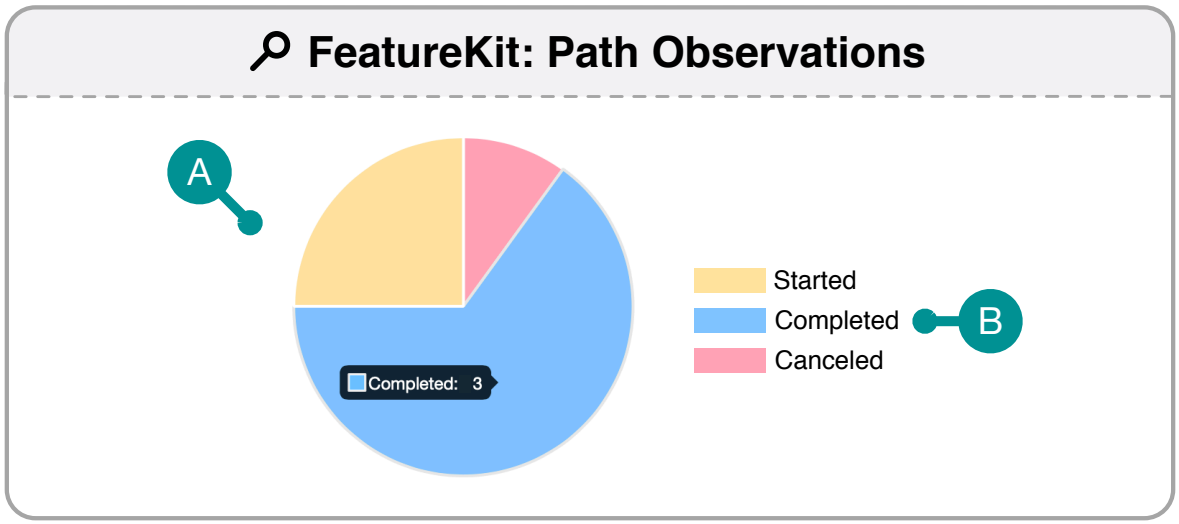


Figure 8.2: Sketch of a spot widget for FeatureKit usage knowledge.

To make use of FeatureKit, developers add two or more feature crumbs to their source code, using the method: `CUU.seed(name:''Crumb Name'')`. Following the maps example (Section 8.1), this might be calling the seed method and passing the parameters *"A: Map View Did Appear"* in the view did appear statement of the view controller, and *"D: Round-circled Button Pressed"* in the action handler for the button that triggers the layer selection. Hereafter, in order to calculate the feature's state (Figure 7.4), the developers need to define a feature path in the **Cuu** dashboard that uses the same names they applied in the source code. Note that this can be done only as soon as the developer pushes the commit with the source code that includes the feature crumbs, as they need to select the respective commit within the **Cuu** dashboard. After completing these two steps, FeatureKit starts observing and analyzing the feature crumbs in order to derive insights about the feature observations.

We implemented three widgets for FeatureKit; in Figure 8.2, we sketch a spot widget to depict usage knowledge in form of feature path observations.

This widget supports developers in understanding the feature adoption. The pie chart (Ⓐ) represents all users that interacted with the feature, i. e., triggered its first feature crumb. Developers retrieve specific numbers of the feature states (Ⓑ) by hovering over the respective part of the chart. Following the color coding in Figure 8.2, developers strive to achieve a feature implementation that is dominated by blue color, i. e., completed feature paths. Yellow path observations, i. e., started feature paths, indicate that users are either in a state of currently performing the feature, or temporarily stopped using it as they interfered with another feature. Red path observations indicate canceled feature paths, i. e., in case users will clearly no longer be able to successfully finish the feature. The widget enables developers to gain a first impression about the feature performance. To investigate on the reasons, FeatureKit provides two additional widgets which we depict in the Appendix C.2.

In Figure C.2, we sketch a FeatureKit spot widget that displays a list of all feature crumbs that were detected for the selected release, including the number of observations, i. e., how often they were triggered by the user. Following the maps example (Section 8.1), the widget clearly shows that there were six occurrences in which the users were likely to start using the feature, as they triggered the first feature crumb *"A: Map View Did Appear"*. However, none of them triggered the subsequent feature crumb *"D: Round-circled Button Pressed"*. This provides the developers with an indication of where to start improving their feature.

In Figure C.3, we sketch another FeatureKit spot widget that highlights how usage knowledge augmented with another usage information can provide more insights on the feature under development. The widget lists unique device types in relation to the feature path observation. Figure C.3-Ⓒ suggests that there might be a problem for users in finding the round-circled button in small screen devices, such as the iPhone X, in comparison to tablet sizes, such as the iPad Pro.

InteractionKit

InteractionKit is a knowledge source that is concerned with monitoring a variety of usage data that is produced during the interaction of the user with the application at hand. This includes data such as device or application events, motion data of the device, as well as touch and gesture events. InteractionKit is part of the **Cuu** SDK[17]. In Figure 8.3, we sketch a spot widget that provides access to InteractionKit data.

The monitored usage data by InteractionKit strives to fulfill two purposes: on the one hand, it can be used by other knowledge sources, i. e., as an input source for classifiers or to enrich their own data source. On the other hand, the data stream by InteractionKit can serve as a detailed protocol of the users' interactions which may become relevant if a particular situation should be investigated. We refer to this protocol as a usage data trace. During the inspection of such a trace, usage data from other sources may be relevant as well, which is why the spot widget in Figure 8.3 is not limited to InteractionKit usage data.

A usage data trace is always associated with a session Ⓐ, which reflects a temporal range that is defined by the beginning and the end of an interaction by an individual user. After the developer selected a session, they are presented with a list of usage data records in an ascending time order. Multiple knowledge sources can post their usage data into this list; their data can be de- and activated using the buttons at the bottom of the widget (Ⓒ). In case a knowledge source provides a fine-grained hierarchy of usage data, the individual entires can be further specified (Ⓓ). This selection process also allows developers to keep track of relevant usage data; in particular as InteractionKit supplies many usage data during each session.

[17]InteractionKit was developed as an open source Apple iOS framework. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/InteractionKit.

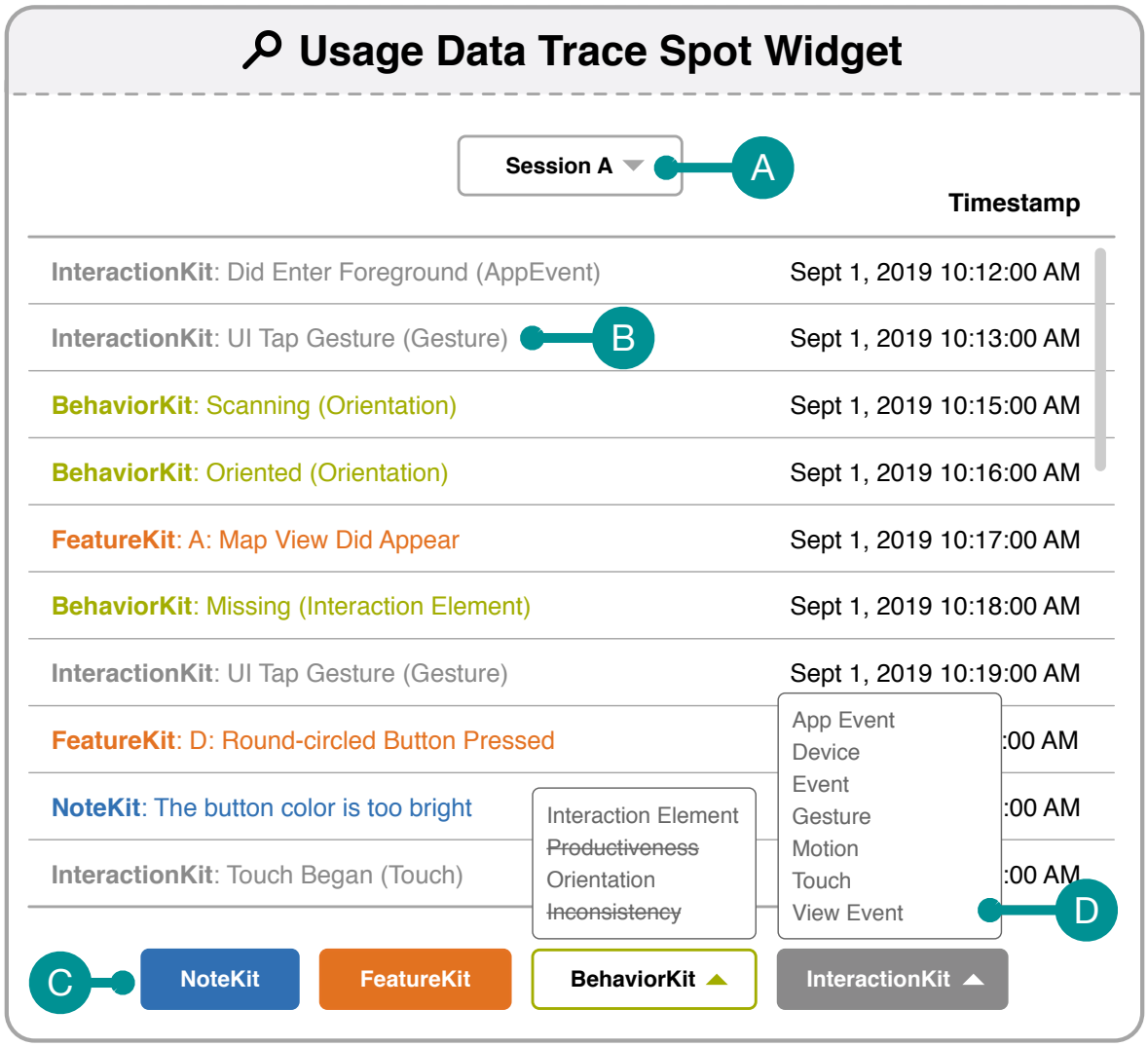


Figure 8.3: Sketch of a spot widget for different types of usage data.

The developers can further inspect a usage data entry by clicking on its list entry (Ⓑ), for example to understand which gesture a user applied, how long the gesture lasted, or where its start and end position are located on the screen.

NoteKit

NoteKit is a knowledge source that enables explicit user feedback collection. It allows users to push spontaneous feedback to the developers. By shaking the device, users trigger an alert that provides a single-line text input field. While the gesture allows them to provide feedback at any time during application usage, the limitation of the text field enforces short feedback and emphasizes the role of NoteKit as a supportive knowledge source for **Cuu**^{SE}. NoteKit is part of the **Cuu** SDK[18].

[18]NoteKit was developed as an open source Apple iOS framework. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/NoteKit.

In Figure C.4, we sketch a spot widget that visualizes user feedback collected using NoteKit. It contains a list of the feedback with the respective timestamp. The example user feedback that is provided in the context of the map example (Section 8.1 highlights the range of user feedback and how it can support the developers in their work. First, *"I am looking for the options to change the map layer"* (Ⓐ), represents a valuable insight to developers: it can be categorized as usage knowledge, as it contains useful information that relates to a feature. The judgement that is implied by the fact that the user is searching for the button describes the need for action for the developer. Second, *"The button color is too bright"* (Ⓑ), does reflect a valuable insight to the developer, however, there is no reference which button the user refers to. While this makes the feedback less useful, the timestamp may provide an indication for the developer to start their investigation, using the usage data trace view in Figure 8.3. Third, *"I like it"* (Ⓒ), may be understood as a positive feedback that is lacking even more of a reference than the previous example and may be skipped in favor of other feedback that can be utilized with less interpretation effort.

8.2 Emotion Extraction from Facial Expressions

The study of person-related characteristics using hardware sensors reflects a broad research field, for example when assessing the cognitive workload during the performance of mentally demanding tasks [173]. We found that consumer hardware may be used as a viable means to derive similar insights [272]. In particular the use of recognition techniques in the wild has been in the focus of research: For example, Heinisch *et al.* showed that physical activities tend to be unlikely to interfere with emotion recognition models [137].

Behavioral data by users support the elicitation of requirements [195]. Biometric measurements are useful to determine emotional awareness [111]; the emotional response can be utilized for user interface assessment [49]. Facial expressions can form another measurement to derive user emotions and thereby reflect an implicit user feedback source. Previous approaches for facial expressions relied on external, stationary equipment [318] that hinders their application in the everyday life of users. The availability of current infrared three-dimensional (3D) cameras made face recognition technology accessible to a growing audience of users in their day-to-day life[19]. This advancement allows deriving user emotions *in-situ* on mobile devices in their target environment.

Based on these assumptions, we developed *EmotionKit*, a knowledge source that is able to process user emotions from facial expressions and to relate them to user interface events. It harnesses consumer hardware camera technology in order to ex-

[19] Apple Support Documentation: *About Face ID advanced technology*. November 2018. Available online: https://support.apple.com/en-us/HT208108.

tract facial features. EmotionKit does not require a machine learning approach to continuously calculate emotional measurements, as it relies on a list of common facial expressions [90, 266]. EmotionKit is based on the assumption that users' current feelings, expressed by their emotions, relate to their current activity, such as interacting with a user interface. This knowledge can contribute to the identification of problems in interactive mobile applications, without the need of complex user studies or domain experts.

In the following, we detail technical aspects of EmotionKit, i. e., how to detect and report emotions using the smartphone's front-facing camera. We sketch a spot widget for user feedback that is provided by EmotionKit. We furthermore detail insights into the foundations of emotions as well as present related work. As part of the treatment validation in Part IV, we present an in-depth investigation of the quantitative and qualitative results of EmotionKit: In a laboratory user study with 12 participants, we evaluated the applicability of EmotionKit and investigated relationships between observed emotions and usability problems in interactive mobile applications. We detail the results in Chapter 10.

8.2.1 EmotionKit

EmotionKit is a knowledge source that converts facial expressions into emotions. EmotionKit is part of the **Cuu** SDK[20]. In the following, we provide more insights on the foundations of emotion research, describe implementation details of EmotionKit, and present a spot widget to visualize its usage knowledge.

Emotion Foundations

Emotions are important for understanding human intelligence, decision-making, and social interaction [246]. The emotion literature is classified into two theories with respect to the term *emotion*: On the one hand, basic emotions refer to a list of the terms anger, disgust, fear, joy, sadness, and surprise [54, 91]. On the other hand, the dimensional theory describes emotions using multiple dimensions [275, 315]. *Activation*, *valence*, and *ambition* are the most commonly used dimensions [276]. EmotionKit builds upon the basic emotions theory. Humans manifest their emotions in implicit and explicit forms such as facial expressions, speech, or body language [4, 92, 193]. In addition, to a certain degree, the expression of emotions is specific to other aspect such as the spoken language and culture [4].

Changes in facial expressions result in different facial features. They can be described using the Facial Action Coding System (FACS), which encompasses a set of

[20]EmotionKit was developed as an open source Apple iOS framework as part of research projects in collaboration with Jan Philip Bernius [33, 34]. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/EmotionKit.

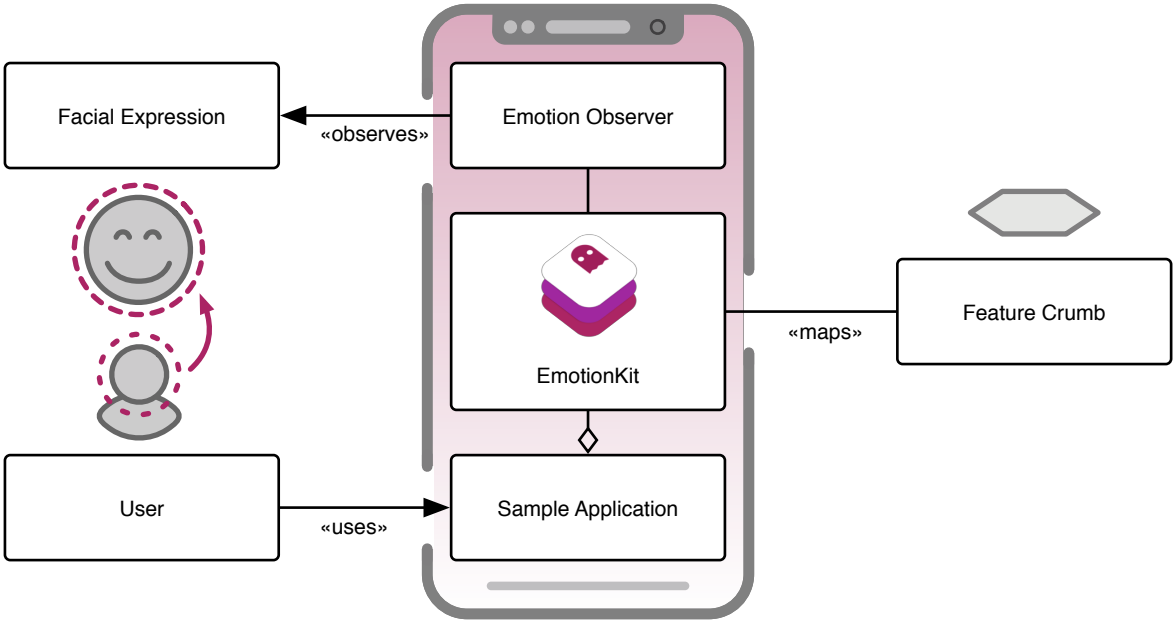


Figure 8.4: High-level overview of EmotionKit, adapted from Johanssen *et al.* [151] (UML class diagram, © 2019 IEEE).

44 Action Units (AU) that define a state of facial features or contraction of facial muscles [90, 193, 238]. It allows the description of thousands of possible facial expressions as a set of a few AUs [321]. The judgment of human emotions can be based on the examination of facial behavior by observers [91]. However, edge cases complicate the detection of emotions; the analysis of muscle movement over time is required, rather than the inspection of a capture of a point in time [92]. Facial expressions identification enables the mapping of a set of AUs to emotions. A system defining such a mapping is the Emotional Facial Action Coding System (EMFACS) [192, 193]. Most people cannot control all of their facial expressions [71].

Implementation Details

Figure 8.4 sketches a high-level overview of the prototypical implementation.

The EmotionKit leverages Apple's ARKit. It utilizes an emotion observer that is built into the smartphone. This allows the observation of the users' facial expressions while they are facing the application for interaction. Using the knowledge extracted from the recognized emotions, it enables the mapping of usability problems to the software version that is currently used by the user.

The implementation of EmotionKit follows a funnel logic, as illustrated in Figure 8.5. The figure shows how the so-called *BlendShapeLocation*s 24, 25, 48, and 49 are translated to the AUs 6 and 12, while these in turn relate to emotion 1, which may be *happiness*. We use this approach to reduce ARKit measurements to emotions, a process which is separated into three major stages.

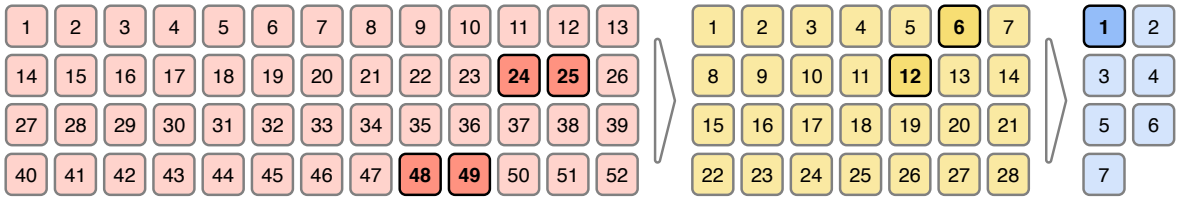


Figure 8.5: Funnel logic to transform ARKit *BlendShapeLocation*s to emotions.

Extracting facial expressions from ARKit During the first stage, facial data is collected using a *TrueDepth* camera. This camera type is available in new generation smartphones, such as iPhone X. ARKit is in charge of performing face detection and extraction of facial features. Then, the data are made available by implementing the `ARSessionDelegate` in the applications' `ViewController`s. We use the *ARAnchors* to retrieve facial data for every face that was detected. These data are stored and passed on to the next stage.

Conversion to FACS Action Units During the second stage, the ARKit data, the *BlendShapeLocation*s[21], is translated into AUs. We summarize the details of the mapping in Appendix C.3 in Table C.1: One AU can be defined by one, one out of two, or the average of many *BlendShapeLocation* values. This reduces the number of data points from 52 down to 22. Notice that FACS defines 28 AUs, however, due to ARKit limitations, we can only detect 22 AUs. While Apple does not refer to AUs in the online documentation, our mapping as described in Table C.1 in Appendix C.3 suggests that their *BlendShapeLocation*s are inspired by the FACS system. Meanwhile, Apple makes a distinction between left and right face movements, which is why we decided to return the stronger sides shaping in our implementation for most of the AUs conversions. Three AUs are described best by a combination of two shapes; this is the reason why we use their average.

Conversion to Emotions following EMFACS In the last stage, we use the EMFACS system to map action units to emotions. It describes each emotion in terms of AUs [192]. Given this relationship and the AUs that were computed in the previous stage, we are now able to derive probability values for each emotion. As we follow this universal definition of emotions, we neither require a training step in advance, nor computation intensive classification tasks. Note that ARKit might do these kind of operations, however, in an energy efficient and operation system-controlled way. EmotionKit calculates the averages for all AUs. Eventually, we can translate each face in probabilities for the emotions defined within the EMFACS system.

[21] https://developer.apple.com/documentation/arkit/arfaceanchor.blendshapelocation

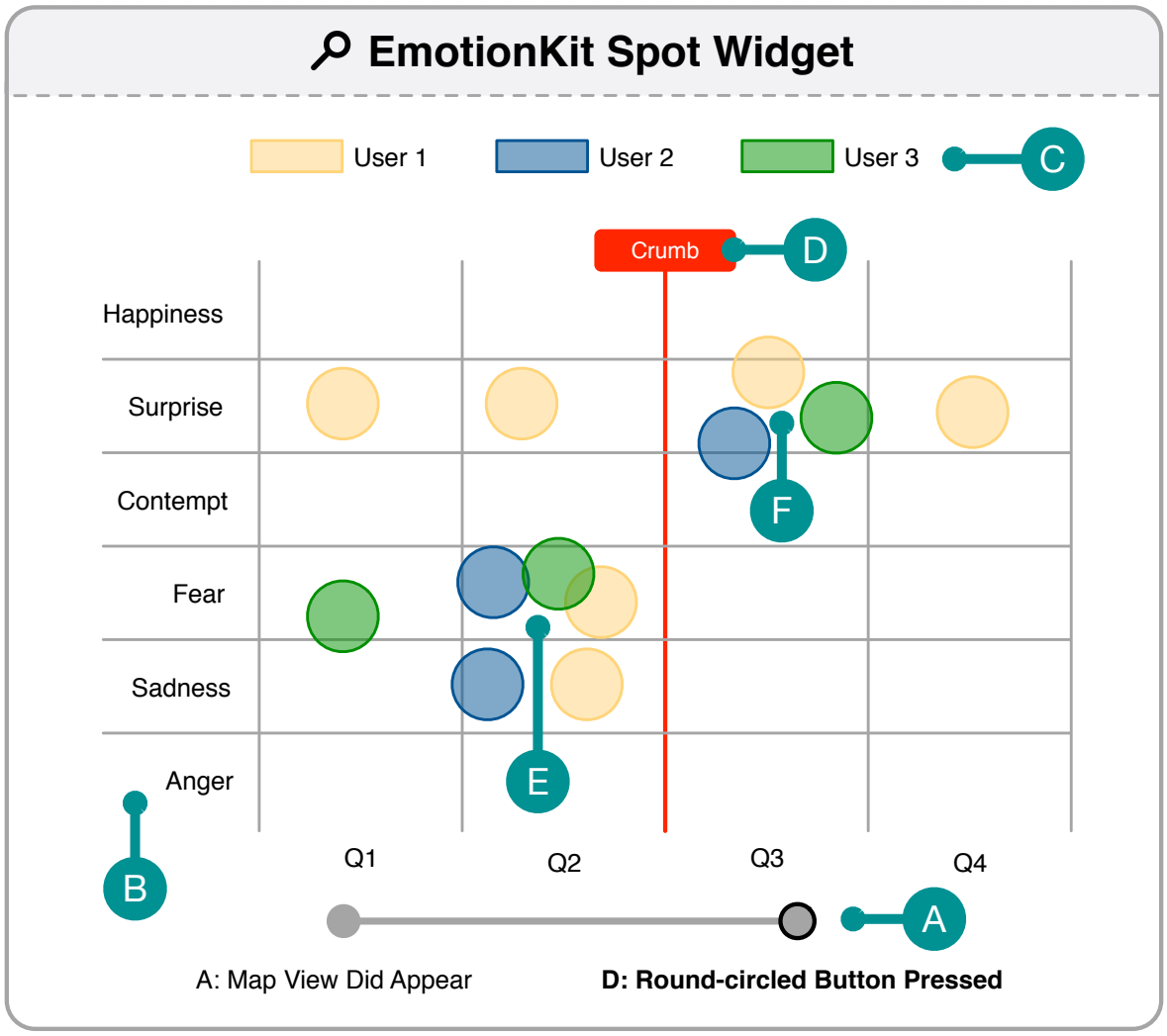


Figure 8.6: Sketch of a spot widget for EmotionKit usage knowledge.

Usage Knowledge Spot Widget

Besides the ability to monitor and process facial expressions, as well as relating the derived emotions to a feature crumb, EmotionKit also provides a visual representation of the usage knowledge: Figure 8.6 presents a spot widget for EmotionKit usage knowledge.

Visualizing emotional data is a complex task and different approaches have been proposed by literature dependent on the context, such as for visualizing emotions of tweets [212] or computer game play [200]. We present a presentation of the high-level results of EmotionKit, the six basic emotions. Given the fact that developers may not be domain experts in the field of emotion identification and interpretation, the main intention of the visualization is for developers to develop a feeling of whether a certain aspect of the application caught the attention of the user in terms of one or more emotions.

We describe Figure 8.6 following the running motivation that we based on the example described in Section 8.1. In order to investigate the emotional responses, the developers first select a feature crumb in the EmotionKit spot widget: All available crumbs for the release are displayed at the bottom of the widget. Since the new round-circled button was the major change of this version, this might be of interest for the developer. A black border and bold typeface indicates that this crumb has been selected for investigation Ⓐ.

The upper part of the widget shows the usage knowledge provided by EmotionKit, separated in a table structure in which the rows reflect the occurrences of one of the six emotions Ⓑ. The emotional responses are depicted per observed user Ⓒ; this is required, as creating average values across their responses would blur their individual emotions.

In the middle of the main canvas of the widget, a vertical red line indicates the point in time of when the selected crumb was triggered Ⓓ. Before and after this mark, the table is divided into four quadrants, each of them summarizing the respective emotion over a time frame of 3 seconds—we detail the aspect of time frames of emotional responses in Section 10.1.3.3. For instance, while *Q2* summarizes the 3 seconds just before the respective crumb was triggered, *Q4* depicts the emotions observed during the time span that lasts between 3 and 6 seconds after the crumb. The summarization of emotions in these quadrants reduces complexity for the developers, in particular as measurements with a low probability are omitted. As developers work with the widget, they look out for a pile of emotional responses, such as in Ⓔ. There, shortly before the crumb has been triggered, multiple users appear to have a strong emotional response. This might indicate that there may be a problem with the user interface or the task that they are trying to achieve. In the example, the response might represent the confusion that users are confronted with in the moment before they found the round-circled button. This may be further supported by a certain relieve which could be derived from the multiple responses of *surprise* in Ⓕ. We suggest to rely on the emotions as a simplified indication for the need to further investigate the particular part of the application. As we further investigate in Section 10.1, the emotion does not necessarily reflect the users' actual emotion, in particular its meaning must be considered in a broader context.

8.2.2 Related Treatments

EmotionKit relies on facial expressions that are generated by existing components, i. e., an observer that is built into a smartphone. In contrast, other work addressing the process of extracting facial expressions on a lower level by starting their investigation on the bare depth information [290, 303]

Deriving and utilizing user emotions that are derived from facial expressions using consumer hardware recently attracted researchers' interest: Scherr *et al.* and

Menning *et al.* [207] propose approaches to utilize the 3D camera and the derived facial expressions in a similar way as EmotionKit. They present a vision of how emotional tracking can be implemented in order to support rapid product changes [207]. In contrast to EmotionKit, they collect a ground truth of emotional data by recording emotional responses from professional actors.

Feijó Filho *et al.* describe a "system to implement emotions logging in automated usability tests for mobile phones" [101]. They further augment the system's capability with additional context information, such as the users' location [99, 100]. As with EmotionKit, they utilize facial expressions to derive insights about the emotions, however, they rely on two-dimensional images that are send to a remote server. There, an emotion recognition software decodes the individual images and tries to detect emotions in the faces. In contrast to this procedure, EmotionKit harnesses facial expressions recorded from a 3D depth camera. Furthermore, EmotionKit performs all calculations on the device; only the result of its processing is send to the **Cuu** workbench. In addition, Feijó Filho *et al.* report on an initial evaluation with two subjects, focusing on positive and negative emotions [99, 100, 101], in which they report a successful application of the system; we present a study with 12 participants including a fine-grained analysis of results. We detail our results in Section 10.1.

OpenFace 2.0 is a toolkit for facial behavior analysis [22]. It is capable to detect multiple face-related features, such as facial landmark detection, head pose estimation, eye gaze estimation, and facial expression recognition. Similar to the processing that is done by EmotionKit, the authors chose AUs as an objective way of describing facial models.

Face-Reader applies a vision based *fun-of-use* measurement [318]. The approach follows the FACS and EMFACS measurements to detect emotions from facial models. FaceReader was evaluated in a controlled test environment, in which subjects were video-taped and measurements compared with researchers' observations. The system detects minor changes not noticed by the manual observation. This supports the underlying assumption of EmotionKit that it is possible to detect subtle changes in human reactions that might be unrecognized by human observers.

The analysis of facial expressions is used for different purposes. McDuff *et al.* describe a system for automatically recognizing facial expressions of online advertisement viewers [206]. They analyzed the response to internet advertisement videos. The goal of the experiment is to judge whether the user appeals to the advertisement and eventually to predict their purchase intend [206]. Viewers' faces were automatically feature coded following the FACS system, from which emotions were derived. In contrast to EmotionKit, McDuff *et al.* analyze the users response to a content, i. e., the advertisement videos. We discuss the differences in responses toward user interfaces, i. e., interactive views, and content-related impressions, i. e., static views, in Section 10.1 . In addition, rather than relying on the data stream from a webcam,

we base our emotion recognition on the output of a 3D camera.

Emotion recognition on consumer hardware becomes more available with commercial frameworks and services for both on-device[22] and online[23] emotion recognition. In contrast to EmotionKit, these frameworks and platforms process and evaluate two-dimensional images to detect users' emotions; in addition, they rely on previously trained data and on an online server, respectively.

The availability of 3D cameras in consumer devices attracted the attention of developers. For example, *Loki*[24] was developed as a proof-of-concept for emotion-targeted content delivery, a similar intention to the work presented by McDuff *et al.* [206]. The developers use the same input parameters used by EmotionKit, i. e., *BlendShapeLocation* provided by ARKit. However, they process diverges starting after the first processing step of EmotionKit described in Section 8.2.1.2: In order to derive the emotions from the facial expressions, they rely on a machine learning approach to derive the emotions from the facial features. This ended in manual coding efforts for which the developers report as a major challenge to solve.[25] In contrast, by relying on EMFACS, EmotionKit uses an established psychological concept to derive the users' emotions in mobile applications without the need for an initial training process.

8.3 Feedback Analysis from Verbalized Thoughts

The Thinking Aloud method represents a value source for explicit user feedback that promises the retrieval of expressive insights: Jakob Nielsen describes it as the *discount usability engineering* method since it allows users to immediately verbalize their thoughts while using an application [216]. The access to such a rich and qualitative feedback source supports the developers during software development, in particular with respect to improving the usability of an application. However, the application of Thinking Aloud during CSE faces challenges.

First, since CSE is characterized by frequent and multiple changes, it is of particular importance that the *spontaneous* impressions of users' are transmitted to developers in a "cheap, fast, and easy" manner [278]. Traditional Thinking Aloud impedes this goal: in an extreme form, it requires users to attend a dedicated test setup in a laboratory environment and physically sit together with the developer.

Second, it has been acknowledged that stopping processes frequently to perform

[22] https://www.affectiva.com/product/emotion-sdk/

[23] https://azure.microsoft.com/en-us/services/cognitive-services/emotion/

[24] https://github.com/nwhacks-loki/loki

[25] *"One of the challenges we ran into was the problem of converting the raw facial data into emotions. Since there are 51 distinct data points returned by the API, it would have been difficult to manually encode notions of different emotions. However, using our machine learning pipeline, we were able to solve this."*—Report of the developers. Available online at https://devpost.com/software/loki-fnbl1d.

regression testing accounts for a major waste of development time [267]. This effort will most likely be further increased if developers are repeatedly performing and analyzing Thinking Aloud protocols, as this requires many manual process steps.

We argue that a successful integration of Thinking Aloud in CSE requires a scalable approach that reduces manual activities. This decreases the time and effort investment for users and developers. On the one hand, users would be able to verbalize thoughts in their target environment without the need to have a developer sitting next to them. On the other hand, developers would be able to focus on the utilization of user feedback, rather than on its collection or processing.

We developed *ThinkingAloudKit*, a knowledge source that enables the Continuous Thinking Aloud (CTA) approach. CTA automates the Nielsen's Thinking Aloud method to obtain better insights into users' thoughts on a new feature increment. ThinkingAloudKit asks users to record speech feedback for a new feature increment as soon as they start interacting with it. These recordings are then analyzed automatically on a sentence level basis. Therefore, we use a natural language classifier to distinguish the content of the user feedback and categorize it into one of four categories of sentiment, namely an insecure, neutral, positive, and negative sentiment. The results of the analysis are visualized in accordance with the new functionality, i. e., a feature crumb, within a widget. This provides developers with a powerful tool to efficiently utilize and benefit from expressive user feedback during CSE.

ThinkingAloudKit achieves an advancement given the *continuous* aspect that leads to additional benefits as compared to the traditional Thinking Aloud method. In particular the fact that the environment where users perform the CTA sessions is the actual target environment—a CSE principle which is enabled by continuous delivery. This increases the likelihood for users to provide additional and more relevant feedback. In addition, a potential bias introduced by a supervisor who is constantly observing the user may be removed. While Nielsen advices that this should be avoided [216], Hertzum *et al.* indicate that moderated and unmoderated thinking aloud tend to provide the same verbalizations [140].

8.3.1 ThinkingAloudKit

ThinkingAloudKit is a knowledge source that automates the Thinking Aloud protocol by recording, transcribing, and analyzing verbal user feedback on a sentence level. It proposes an approach for continuous feedback collection. ThinkingAloudKit is part of the **Cuu** SDK[26]. In the following, we provide more insights on the foundations of the Thinking Aloud protocol, describe implementation details of ThinkingAloudKit, and present a spot widget to visualize its usage knowledge.

[26]ThinkingAloudKit was developed as an open source Apple iOS framework as part of a research project in collaboration with Lara Marie Reimer [254]. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/ThinkingAloudKit.

Thinking Aloud Foundations

Thinking Aloud originates from psychological research with the goal of developing a method to model cognitive processes: By inspecting the written transcripts of the verbal explanations of subjects, different approaches in solving a problem become identifiable [306].

The method's idea was transferred to software engineering as a means to primarily support the identification of usability problems through asking subjects to provide feedback on what they see and what they do while using the software [191]. Nielsen describes Thinking Aloud as "may[be] the single most valuable usability engineering method" [216], as it reflects users' inner thoughts about a software which they would refrain from providing during normal application usage. He emphasizes that the main strength of the method lies on the quality of feedback gained by only a few testers; notably, such feedback may have been biased based on the testers' subjective theories regarding the software, when conducting a Thinking Aloud session. Since Thinking Aloud reveals the underlying reasons for the user's actions [145], it allows developers to gain a better understanding of application usage and to detect usability issues they might not have known before.

We refer to Nielsen's description of Thinking Aloud [216] as the *traditional* method, since a tester and a supervisor physically sit together with the former talking while using the application, whereas the supervisor is taking notes which are used later to manually identify usability problems.

Implementation Details

The CTA approach adapts the traditional Thinking Aloud method to an automated setting that builds upon frequent and rapid software increments. Figure 8.7 outlines the main actions and control flows that are require to complete a CTA session. Using four perspectives, we identify three actors: First, the developer who is in charge of setting up the CTA approach and is later concerned with its analysis. Second, the user of an application. Third, the ThinkingAloudKit. Except for the setup, all activities are repeated multiple times per feature increment, without the need for further manual interaction by the developer. This is indicated in Figure 8.7 through the multiple boxes behind the respective set of actions.

Developer Perspective: CTA Setup To activate a feature under development for CTA, the developer starts to *Prepare [a] Feature under Development* by adding *Feature Crumbs*, for example, to reflect button actions or new screen view events. Then, the developer needs to *Define [a] Feature Path in [the]* ***Cuu*** *[D]ashboard*. The result of this activity is a new feature path version that consists of a concatenation of multiple feature crumbs that describe a linear sequence of a feature. The feature version and

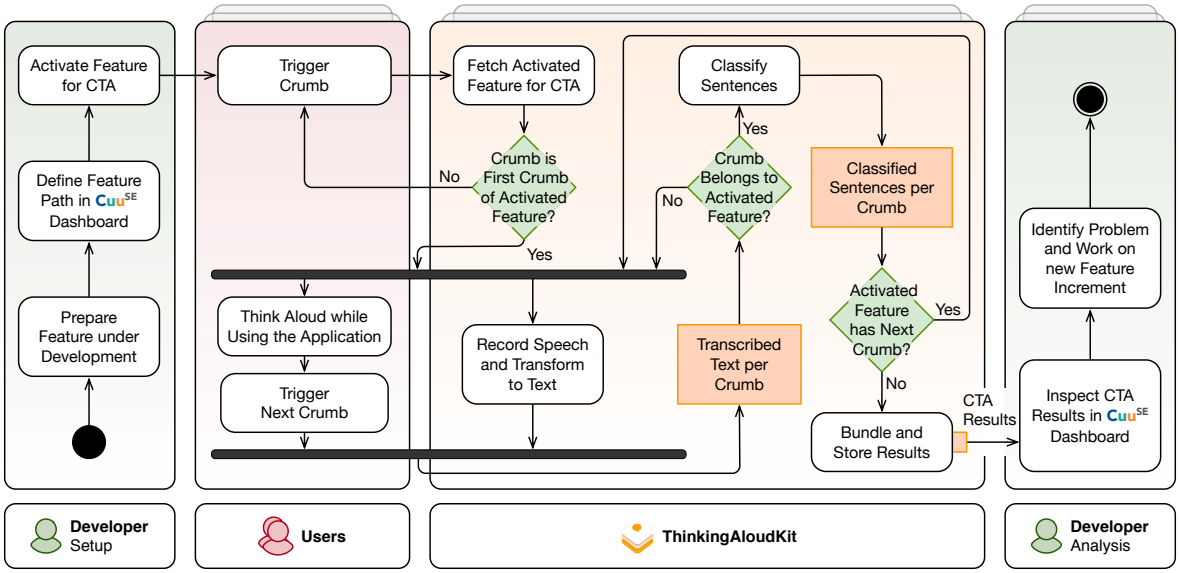


Figure 8.7: Continuous Thinking Aloud, adapted from Johanssen *et al.* [159] (UML activity diagram, © 2019 IEEE).

the related feature crumbs are defined once per commit. After the feature crumbs were added to the commit and the feature path is defined in **Cuu**, the developer needs to *Activate [the] Feature for* CTA; this is also done in the **Cuu** dashboard and enforces the interactive capabilities of the dashboard.

User Perspective Application users, i. e., testers or end user, access a new release that contains a feature under development and start using it. As soon as they *Trigger [a Feature] Crumb* that is part of the previously defined feature by the developer, they are asked whether they would want to start participating in a CTA session. In Figure 8.7, we omitted user interface-related activities that are triggered by CTA in the application to maintain the readability of the activity diagram.

If the user opts out, feature crumbs will be ignored for the current session and no data is recorded. However, if they agree, ThinkingAloudKit will start recording their *Think[ing] Aloud while [they are] Using the Application*. A screen initially presented to inform users about the CTA session contains a reference to the feature that was detected. It also further provides suggestions for the type of speech feedback expected, given that the user may be unfamiliarity with concept of Thinking Aloud.

Afterward, the users continue to *Trigger [the] Next [Feature] Crumb*, while they use the application and verbalize their thoughts. After ThinkingAloudKit determines that the last feature crumb has been triggered, an information screen is shown to the users and the recording is stopped. This is the end of the interaction with the user.

ThinkingAloudKit Perspective The ThinkingAloudKit performs the core actions for automating the traditional Thinking Aloud method to enable the CTA approach.

After a user initially triggers a feature crumb, the ThinkingAloudKit *Fetch[es the] Activated Feature for* CTA from a remote server that hosts all features that were previously flagged, i. e., activated, by the developer. ThinkingAloudKit initiates the CTA process in case the *Crumb is [the] First Crumbs of [the] Feature*. Subsequently, CTA continuously *Records Speech and Transform[s it] to Text*. Whenever a new feature crumb is triggered and the *Crumb Belongs to [the] Activated Feature*, the resulting text are transmitted to a natural language classifier. If this is not the case, CTA proceeds to record speech and awaits the detection of the next feature crumb. On a sentence level, the classifier determines whether the text can be categorized containing either an insecure, neutral, positive-, or negative sentiment. Given the limitation of the available data set as described in Section 10.2, we restricted the classifier to four categories. A more fine-grained differentiation may result in a better analysis process. We detail categories in the following:

- **Insecure** sentences express that users are unsure of what they should do. Sentences that contain an insecure sentiment often contain words such as "maybe" or verb forms such as the conjunctive.
- **Neutral** sentences reflect both impressions and actions performed by the user. Impressions describe sentences that reflect the users' visual experience while they use the application. This may include sentences such as "All right, now here I see a picture of a car and a button." Actions refer to sentences that reflect the users' interactions within the application. This may include sentences such as "Now I am going to tap on the button."
- **Positive** and **Negative** sentences form the most important sentiment, since they assess the current state of the application from the perspective of the user. For example, a sentences such as "I love the design of this date picker; it is very simple and intuitive!" expresses a positive sentiment, while "This button is useless; I do not understand what it is good for" indicates a negative attitude regarding the user interface.

This procedure is repeated as long as the *Feature has [a] Next Crumb*: In case the feature still consists of more feature crumbs, CTA continues to record the speech. Otherwise, the *Classified Sentences per Crumb* are *Bundle[d] and Store[d]* by ThinkingAloudKit. This also includes the transmission of results to the developer.

Developer Perspective: CTA Analysis After a CTA session becomes available, the developer can *Inspect [the]* CTA *Results in [the]* ***Cuu*** *Dashboard*. This is done using a ThinkingAloudKit widget, which we detail in the next Section 8.3.1.3. Based on the visualization in the widget, they can *Identify [the] Problem and Work on [a] new Feature Increment* to resolve issues in future releases. This marks the end of the CTA approach.

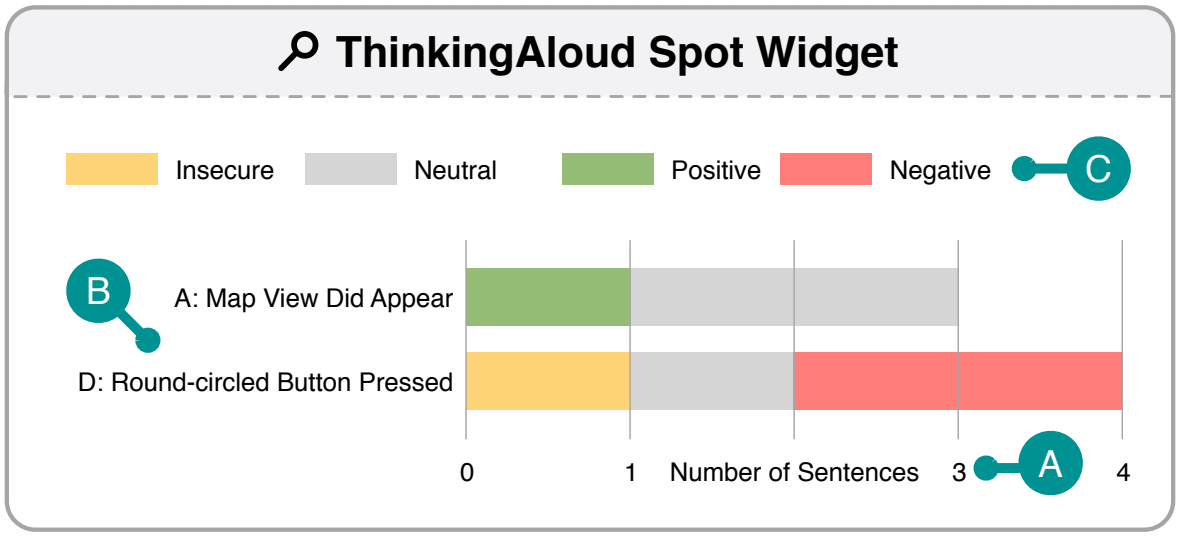


Figure 8.8: Sketch of a spot widget for ThinkingAloudKit usage knowledge.

Usage Knowledge Spot Widget

Besides the ability to record, transcribe, and categorize verbal feedback, ThinkingAloudKit also provides a visual representation of the usage knowledge: Figure 8.8 presents a spot widget for ThinkingAloudKit usage knowledge.

We follow the same example to describe the use of the widget as in Section 8.2.1.3. In contrast to the EmotionKit widget, the ThinkingAloudKit widget does not require the user to select a specific feature crumb for user feedback analysis. Furthermore, the widgets present an aggregation across all users that provided feedback.

The widget in Figure 8.8 presents a bar graph of classification results separated by the feature crumbs that are part of the commit with the feature path v1. The x-axis shows the number of classification results (Ⓐ) with the stacked result of classifications (Ⓒ). The y-axis lists all feature crumbs of the feature path (Ⓑ). Both feature crumbs listed in Figure 8.8 show a total of three instances of recorded sentences that were classified with a *neutral* sentiment. In fact, we expect the neutral sentiment to be the most prevalent, since it is encouraged by Thinking Aloud, i. e., describing what one sees. However, for the most cases, these sentences can be omitted, as they only describe actions and interactions that are performed by users. For the first crumb *A: Map View Did Appear*, we notice one sentence that was classified to contain a positive sentiment; this supports a developer's hypothesis that this particular aspect of the application works as expected and there is no need to invest time and efforts into its improvement. On the other hand, for the second crumb *D: Round-circled Button Pressed*, we observe one sentence that is classified as insecure and two sentence that appear to express a negative user feedback. In the case of the example shown in Figure 1.1, this signals the developers that the newly introduced rounded button to switch the maps layer appears to need an improvement.

8.3.2 Related Treatments

To the best of our knowledge, there exists no work that automates Thinking Aloud during CSE. Since traditional usability testing exhibits low coverage of tested features due to their mostly expensive application [147], *Remote Usability Testing* became an established usability engineering approach, which can be further divided into *synchronous* and *asynchronous* approaches [279]. While during synchronous testing both the user and supervisor may be geographically separated, they communicate during the test in such a way that the supervisor observes and evaluates the user's behavior, e. g., through a virtual three-dimensional laboratory [58]. Asynchronous testing removes the need for a dedicated supervisor which results in time savings. ThinkingAloudKit classifies as a remote asynchronous testing approach. Therefore, it also leverages the time benefits, while at the same time it overcomes the limitations of asynchronous testing that are highlighted by Bruun *et al.* [48].

Marsh *et al.* describe an approach in which they combine speech recordings with captured screen recordings in order to provide a greater context [203]. In contrast to their approach, ThinkingAloudKit realizes the context information by relying on feature crumbs. This enables an even faster analysis of speech results. Furthermore, it enables a better comparability between commits, as the crumbs can be reused.

Guzman *et al.* visualize user feedback with a focus on user emotions [130]. While we outline how we collect and visualize emotions derived from EmotionKit in Section 8.2.1, we want to highlight that ThinkingAloudKit uses similar approaches for both processing text-based feedback and the visualization of the results. However, ThinkingAloudKit focuses more on qualitative aspects of feature increments and thereby addresses the high frequency of CSE.

Ferre *et al.* describe the extension of an analytics framework that enables the recording of user interactions for usability evaluation [104]. ThinkingAloudKit relies on a similar concept to describe a task under investigation, as it uses feature crumbs as the main reference point. However, while Ferre *et al.* focus on usage logs analysis, ThinkingAloudKit additionally utilizes speech to identify usability defects.

8.4 Classification of Individual Users from Usage Data

Following the example in Section 1.1, the usage of the *map layer alter* feature may have declined because of the change of the button position in version 2. This insight could be automatically detected based on usage data on how the users interact with the application. Developers require this information to support their next decision and to judge the changes that were introduced with the latest version. However, most of the time, little is known about users and user feedback lacks contextual information [182] that would help developers [78]. The monitoring represents an un-

obtrusive way of collecting implicit usage data. Then, the inference of user-, usage-, and application-related information helps to close a knowledge gap.

We developed *BehaviorKit*, a knowledge sources that explores the possibilities for externalizing usage information from the behavioral usage data. It reuses the usage data provided by InteractionKit, as introduced in Section 8.1.2.2, and forwards it to individual classifiers that infer concrete attributes regarding users, their behavior within the application, and the application itself, as described in the following.

First, BehaviorKit assumes that user-related information can be inferred from usage data. User-related information refers to immanent characteristics of an individual who is using the application. This supports other knowledge sources in enriching their data, such as PersonaKit that will be introduced in Section 8.5. We expect that the user-related information will not change during an interaction session. We created classifiers to infer the users' age group and smartphone experience.

Second, BehaviorKit assumes that usage-related information can be inferred from usage data. Usage-related information describes what a user is doing while using the application. Overall, we intend to identify the user's familiarity with the application at hand. The respective classifications may or may not change several times during one interaction session. We created classifiers to infer two characteristics: an *orientation* and *productiveness* state. The orientation describes whether users are *oriented* in the application. Users who are oriented know where to find the feature needed to complete their current task. Users who are not oriented are not familiar with the application and have to search for the feature they need; therefore, they enter a *scanning* state. The productiveness state describes whether a user is currently *working* toward a solution to whichever problem they are trying to solve. There are many reasons why users may not be productively working: they might be looking for a feature (i. e., *scanning*), exploring the application in an attempt to improve their overall knowledge of it (i. e., *exploring*), or testing a specific feature (i. e., *experimenting*)—these examples are not exhaustive. For reasons of practicability in the data collection process, we combined all non-working states into a *not working* class.

Third, BehaviorKit assumes that application-related information can be inferred from usage data. Application-related information refers to characteristics concerning elements of the application itself. Overall, we intend to identify users' behavior in the presence of a situation in which they encounter an unexpected system behavior, such as inconsistencies in the user interface or missing user interaction elements in the user interface. This relates back to the practitioners' statement that problems can occur at any point in time. We created classifiers to infer two states which are at either active or not: If the current usage data indicates problems with the user interface, a *usability issue* is present. A *missing interaction element* state is present if the user is looking for hidden or non-existing features they would like to use. By detecting such issues, developers gain general insight on where to start improvements.

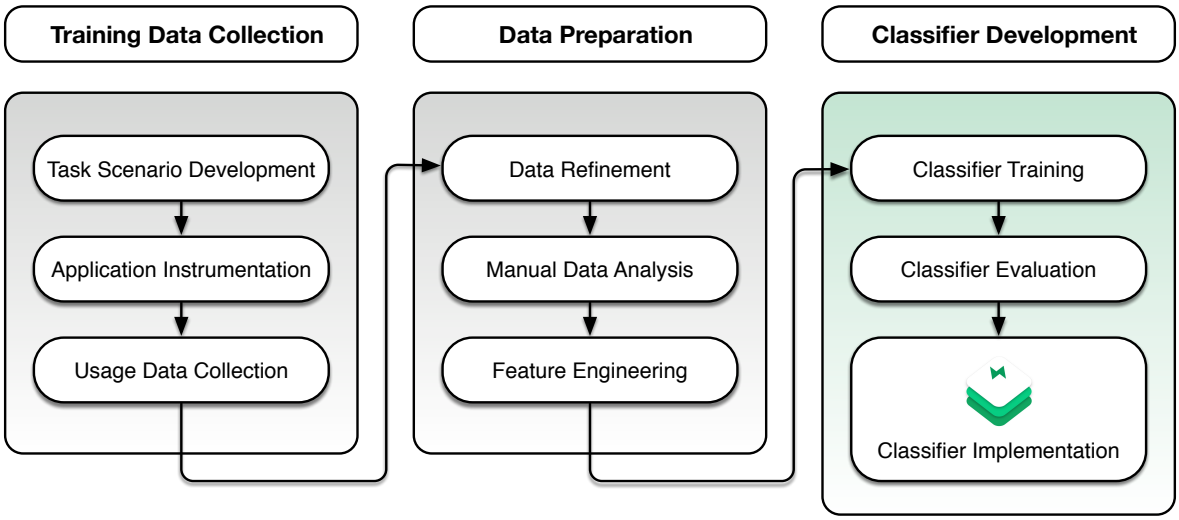


Figure 8.9: The development of BehaviorKit classifiers (UML activity diagram).

8.4.1 BehaviorKit

BehaviorKit is a knowledge source that uses machine learning classifiers to process usage data in form of interaction data into usage information. BehaviorKit is part of the **Cuu** SDK[27]. In the following, we describe implementation details of BehaviorKit, outline how an application can be instrumented for data collection, discuss preliminary classifier results, and present a spot widget to visualize its usage knowledge.

Implementation Details

We divided the development of the BehaviorKit classifiers into three consecutive phases as depicted in Figure 8.9. We detail every phase in the following.

Training Data Collection In the initial training data collection phase, we first developed task scenarios which allowed us to observe a user's state while interacting with a sample application. After instrumenting the sample application, we asked multiple subjects to complete the task scenarios on the prepared application and recorded their behavior to derive labels for the classifiers. For the user-related classifiers, we collected labels in form of *age* and *smartphone experience* through a questionnaire before each data collection session. We detail this aspect of selecting the application and how to instrument it to collect data in more detail in Section 8.4.1.2.

Data Preparation In the data preparation phase, the data was refined for further analysis by removing incomplete or invalid samples. In particular, idle time inter-

[27] BehaviorKit was developed as an open source Apple iOS framework as part of a research project in collaboration with Michael Fröhlich [114]. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/BehaviorKit.

vals, in which participants were asked to answer survey questions, were removed. Then, the collected data was manually inspected and analyzed using WEKA [312]. Finally, feature vectors were generated to capture the monitored user interaction over a window size of 10 to 60 seconds into the past. Following this step, we were able train the classifiers according to the hypotheses based on the refined, raw data.

Classifier Development Eventually, classifiers were developed and implemented for each of the characteristics that BehaviorKit tries to infer. We evaluated different machine learning algorithms sets, i.e., *AdaBoost*, *LinearSVC*, *MLPClassifier*, and *Random Forest*, while also comparing different sets of input features against one another. Using cross-validation, the classifiers were optimized with different subsets of input features. The final classifiers were then evaluated and compared with a test set that was not used for the training to identify the most accurate ones. Classical ensemble techniques, such as *AdaBoost* and *Random Forest*, yielded the most accurate results for most of the hypotheses; The size of the training set did not allow for representational learning approaches, such as neural networks.

Application Instrumentation

The primary goal of the training data collection phase was to obtain sufficient labeled training data for the development of the classifiers. We considered several aspects when choosing the application. Overall, we hypothesize that a reusable classifier can be developed given an approach to collect and label usage data.

Application Requirements The source code of the application under observation must be accessible to instrument it for data collection, i. e., to integrate InteractionKit for the data monitoring. Data collection sessions need to be short; at the same time should simulate real-world usage. This reduces the chances to label noisy data, as identified as one of the challenges in Section 8.1. Participants should be at least vaguely familiar with the tasks they are prompted to complete. The application under observation must be reasonably complex to obtain labeled usage data for all classifiers and should be of production-grade quality as unexpected software faults could reduce the quality of the collected data; this, however, might be not always the case in the environment in which the classifiers will be applied. The application should encompass many well-known interaction elements that are common in mobile applications, such as scroll views, standard buttons, and text input fields. Building upon data produced by common user interface elements, addressing the challenge of frequent changes that lead to different interface (Section 8.1) is mitigated and broad applicability of the usage data should be guaranteed. Given these requirements, we decided to utilize the Wikipedia mobile application for iOS[28].

[28]Source code available online: https://github.com/wikimedia/wikipedia-ios.

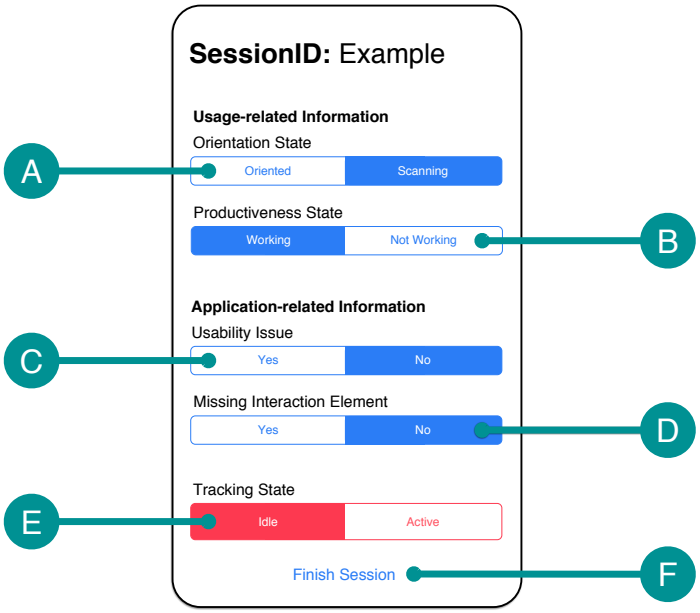


Figure 8.10: Screenshot an annotation application for data labeling.

Data Labeling with Tool Support The creation of labels requires an active observation of what the user is doing. For this purpose we propose tool support in form of an *annotation application*, shown in Figure 8.10.

The annotation application in Figure 8.10 creates a log file that can be aligned with the monitored usage data from the observed sample application. While a subject works on the completion of a given task, an observer selects which labels apply to the respective situation. With respect to the orientation state, the observer tries to answer the question whether the observed user knows where to find and how to trigger the functionality they are asked to use. This can be easily observed since an observer knows what task the user wants to achieve and is familiar with the application, i. e., where the functionality can be found. As a result, the observer toggles the segmented bar in Ⓐ depending on the users' performance. With respect to the productiveness state, the observer tries to answer the question whether the observed user moves toward achieving their goal. This is more challenging as there may be several paths to achieving a specific task. The observer classifies a user as *working* in Ⓑ whenever their actions lead them closer to solving the task at hand. To generate the application-related labels, we modified the sample application under observation and introduced usability issues on certain elements. User interactions with these elements are annotated as usability issues Ⓒ. Similarly, we introduce a missing interaction element by removing or hiding interaction elements, i. e., buttons, that are required to successfully perform the task at hand. When the users encounter such as situation, the observer uses Ⓓ to label the data.

Whenever the task performance is interrupted, e. g., in case of a question or distraction, the observer sets the tracking state to idle Ⓔ to indicate that data collected

Table 8.1: Results of the individual BehaviorKit classifiers.

	Achieved Accuracy (SD)	Majority Class
User-related Information		
Age Group	0.48 (± 0.00)	0.42
Smartphone Experience	0.76 (± 0.05)	0.68
Usage-related Information		
Orientation State	0.83 (± 0.01)	0.60
Productiveness State	0.76 (± 0.00)	0.57
Application-related Information		
Usability Issue	0.79 (± 0.01)	0.65
Missing Interaction Element	0.87 (± 0.00)	0.78

during this time is invalid. This allows to reduce noise in data, which further addresses the challenges described in Section 8.1. Finally, the observer finishes the data recording using the *Finish Session* button Ⓕ at the bottom of the annotation application.

Classifier Results

In this section, we present the results of the individual BehaviorKit classifiers. We created multiple task scenarios and ran each of them with 108 subjects. They reported to be between 18 and 66 years old, with a mean of 30 years; The questionnaire further revealed that 74 of the subjects considered themselves an expert in smartphone usage, while 34 categorized themselves as a novice. On an average, we recorded 9 minutes and 13 seconds of labeled usage data from each subject.

Table 8.1 illustrates the accuracies of the developed classifiers in comparison with the accuracy that one would achieve by guessing the majority class. Overall, the classification accuracy results indicate that some of the classifiers may be applicable for good prediction of the respective information.

The achieved accuracies for identifying the age group were almost as low as the fraction of the majority class. The results indicate that it is not possible to infer the gender and age group using the proposed approach; however, this might be due to the size of the dataset—we address this aspect in Chapter 10. The results for inferring the smartphone experience, i. e., indicating whether a user is more likely to be a novice or an expert user, are better. However, given the high Standard Deviation (SD) of ± 0.05, these results need to be treated with caution.

The classifiers for inferring both usage-related and application-related information are more accurate than guessing the majority class. This allows the assumption

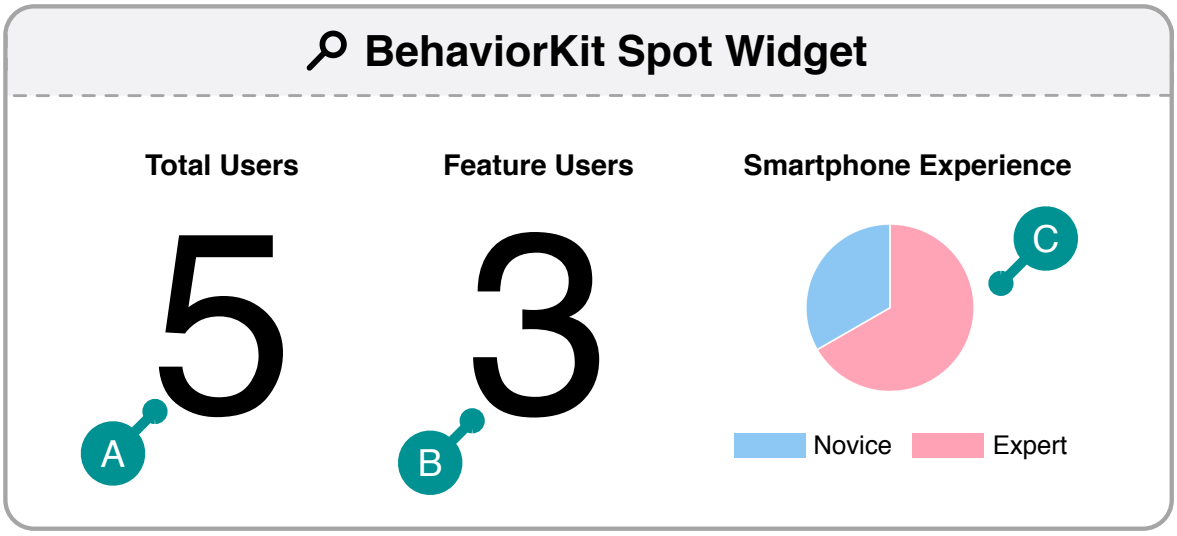


Figure 8.11: Sketch of a spot widget for BehaviorKit usage knowledge.

that the orientation and productive state, as well as whether there exists a usability issue or a missing interaction element, can be detected using the respective BehaviorKit classifier. A comparison of the different input feature sets indicates that *view events* seem to be the most relevant type of information for inferring usage- and application-related information, while for the latter one, *touch events* appear to be an equally relevant factor.

Usage Knowledge Spot Widget

Besides the ability to process usage data, BehaviorKit also provides a visual representation of the usage knowledge: Figure 8.11 presents a spot widget for BehaviorKit usage knowledge.

We decided to combine the usage knowledge that is provided by BehaviorKit with user feedback from other knowledge sources. Thereby, the widget is read from left to right, following a funnel approach: In Figure 8.11, Ⓐ is an insight provided by InteractionKit, while Ⓑ is provided by FeatureKit. In Ⓒ, the results from BehaviorKit's classifier for smartphone experience are visualized as a pie chart. It relates to the number of feature users in Ⓑ.

With respect to the example scenario that we described in Section 8.2.1.3, this may be read as follows: Five users in total were interacting with the version 2; some of them might have just opened the application and quit it, or used it for something else than the feature that is defined in the feature path. Three of the users were completing the feature successfully. The pie chart using the BehaviorKit information reveals that two of them were experts in smartphone usage; this might be interpreted in a way that it is in particular difficult for novice users to find the newly introduced button. Eventually, user information that origins from BehaviorKit is also a valuable

addition to the list view which is part of the usage data traces spot widget.

8.4.2 Related Treatments

Several studies have tried to correlate mobile usage data to user characteristics. Welke *et al.* were able to differentiate users based on the applications they used [309]. Malmi and Weber inferred users' demographics (age, race, and income) from the list of applications that each user had installed [199]. In contrast, BehaviorKit relies solely on data that is produce by the user within the application.

Further research showed that communication patterns can be used to categorize users and infer demographics: Dong *et al.* inferred the gender and age of users based on their communication patterns (calling and text messaging) [84], and Xu *et al.* used network traffic patterns to analyze application usage [317]. In comparison to the approach of Dong *et al.* and [317], BehaviorKit does not use qualitative user data, such as written text, to create the classifications.

The value of inferring information about users from smartphone usage data has already been recognized by other researchers. Teh *et al.* were able to recognize users based on their dynamic touch patterns [299]. Furthermore, several researchers have addressed the challenge of automatically detecting usability issues based on usage behavior: For example, Bader *et al.* developed a heuristic to automatically detect usability issues in mobile applications based on the view presentation log [20]. Similarly, Damevski *et al.* predicted "usage smells" based on sequences of actions using *Visual Studio* [70]. In addition, Schüller developed and evaluated a system for generating user feedback to automatically detect usability issues based on the usage context [281]. BehaviorKit shares the data basis that is used by the presented work; however, it focuses on the development of applications during CSE.

8.5 Classification of User Groups from Usage Data

The development of a feature follows the goal of solving a problem for a user. Goals are elicited based on the users' needs: While the functionality can be described in functional requirements, the need for an easy and simple implementation is depicted in nonfunctional requirement such as usability [216]. In many cases, however, the actual user of a feature cannot be asked, or the individual user may not be empowered to directly influence a solution [64]. For this reason, Alan Cooper introduced the concept of *Personas*, i. e., *"hypothetical archetypes of actual users"* [64], to software engineering and in particular to the early phase of the design process.

Personas refer to a fictional and synthetic character of a user; a version of how one would imagine a user *could* look like, with a focus on certain, relevant, and outstanding characteristics. Personas are typically composed of characteristics, such

as names, demographics, and business-related information and are driven by goals they want to achieve [64]. They are derived from a limited sample of the population to represent a group of real users.

During software engineering, personas support the development of features in case there is only sparse or no explicit feedback available. In particular during the early state of a project, it is important to have archetypical representations of how users use an application, however, sources for the creation of the personas are rare. Traditionally, the creation of personas requires a laboratory environment in which qualitative methods are applied to derive personas for a project upfront, a practice which requires considerable efforts in time and money [43, 250]. In addition, as it applies for most qualitative research methods, there might be a human bias in the creation of the personas that affects the outcome [211]. Overall, the manual activities result in a static perspective on a user group which is built from observations that are not provided by the actual users. This in combination with the disadvantages of the efforts required for the creation as well as the aspect of a human bias limit the applicability of personas in the context of CSE.

We developed *PersonaKit*, a knowledge source that collects monitored usage data and combines occurrences that share similarities. We introduce the concept of *Runtime Personas* [160]. In contrast to the traditional concept of a persona—which relies on the creation of personas by domain experts based qualitative, yet static information—PersonaKit derives runtime personas that reflect a categorization of users based on dynamic aspects that were monitored during their interaction with the application.

PersonaKit extracts a set of unique characteristics from usage data, which is partially derived from InteractionKit, as introduced in Section 8.1.2.2. Based on the data, PersonaKit uses a clustering algorithm to create clusters, i. e., groups, of users that share similar usage behavior. Inspired by the concept of traditional personas and as an additional step to the clustering, PersonaKit extract personality traits of the users and allocates them to the runtime personas. PersonaKit also enriches the derived runtime personas by generating textual descriptions of the collected and summarized usage data, i. e., how the users behaved or in case they used a feature in a particular way. This supports the developer to quickly grasp and understand the groups of users, as the runtime personas can be generated and advanced with every new user and every new feature increment.

In particular with respect to the challenge of transitioning to large volumes of users, PersonaKit becomes an important knowledge sources to distinguish between groups of users and creating a prompt assessment if and how to incorporate other user feedback. Eventually, PersonaKit depicts an additional source of knowledge that is provided to the developer without the need of explicit user feedback provided by the user or the need to manually process data.

8.5.1 PersonaKit

PersonaKit is a knowledge source that uses a clustering algorithm to gather user sessions that share similar characteristics that have been observed from interaction and device data. It enables the creation of basic instances of runtime personas. PersonaKit is part of the **Cuu** SDK[29]. In the following, we provide more insights on the foundations of personas, describe implementation details of PersonaKit, and present a spot widget to visualize its usage knowledge.

Persona Foundations

Personas have been in the focus of other domains, such as for segmentation purpose in marketing [149], before they were adapted by software engineering [64]. Cooper describes the use of personas in case there is no access to the actual end user [64]; which is why the same user attributes should contribute to the description of the persona that also characterize the users that strive for a particular goal. Following Cooper's description, a persona is a byproduct of the result of the investigation process on a user group [64].

The research on personas is wide-spread, in particular with respect to their creation and utilization process during software development. For instance, Ferreira *et al.* propose the use of empathy maps to create personas [105]. Caballero *et al.* found that, in agile development processes, personas are applied as part of one of the following two phases: exploratory and refinement phase [53]. As the result of a semi-structured interview study, Billestrup *et al.* report that practitioners implement their own practices in developing and using persona following their individual understanding—which does not necessarily follow the recommendations of literature [35]. Overall, personas can contribute to the elicitation of requirements [3].

Given the importance of personas for software engineering, the attention of researcher addresses their validation. Faily and Flechais describe an approach that relies on insights from a grounded theory model to augment the empirical data for personas [97]. Salminen *et al.* propose a persona perception scale, which is derived from relevant literature, to provide an instrument for persona validation [269].

Implementation Details

PersonaKit uses quantitative methods to overcome the shortcomings of qualitative methods to create personas. To reflect that the outcome is created from runtime observations observations, we refer to the results as runtime personas. Figure 8.12 details the concept of runtime personas and their creation process.

[29] PersonaKit was developed as an open source Apple iOS framework as part of a research project in collaboration with Florian Fittschen [108]. It is available online: https://github.com/cures-hub/cures-cuu-sdk/tree/master/CUU/Classes/PersonaKit.

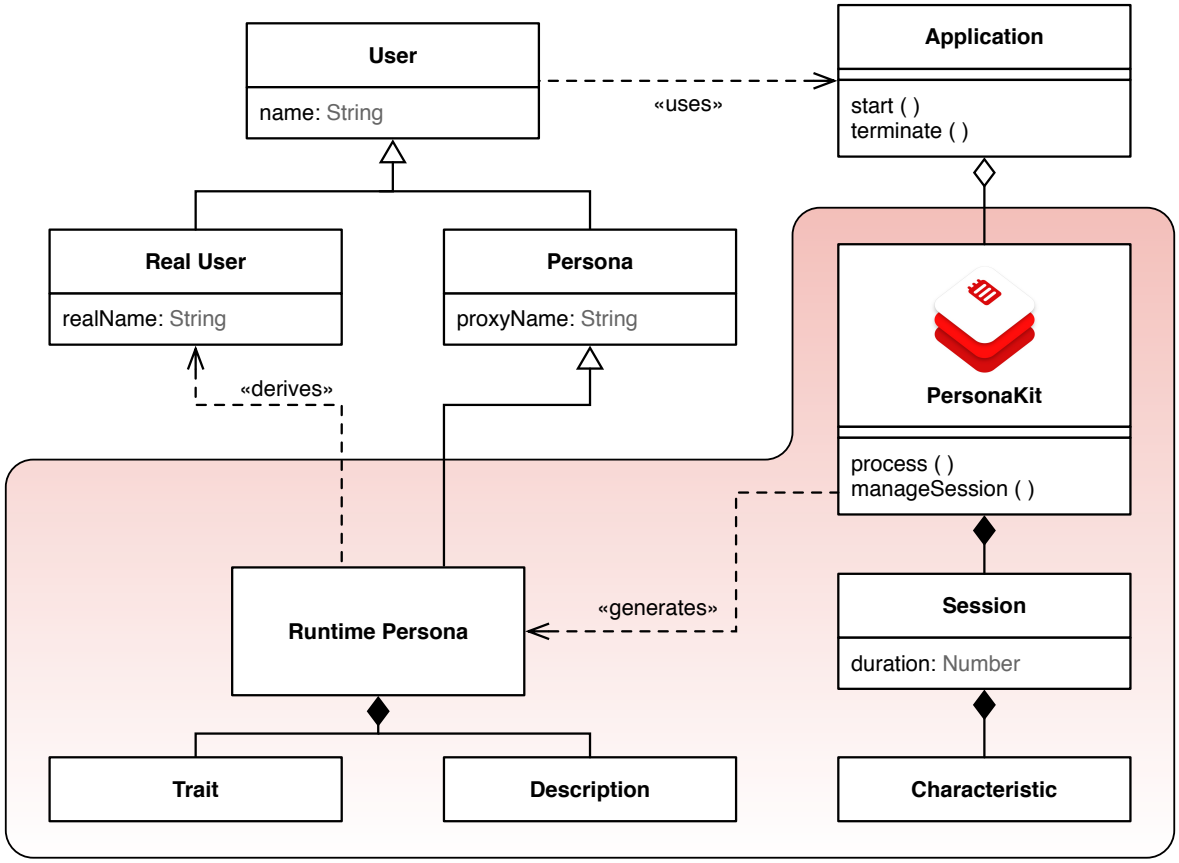


Figure 8.12: Analysis object model for PersonaKit (UML class diagram).

We use the proxy pattern as described by Gamma *et al.* [117] to model the relationship between the superclass `User`, the actual `Real User`, and a `Persona`. The fact that a `Runtime Persona` is a subclass of persona highlights that they share a similar structure, yet runtime personas provide an extension to it in a way that they are built upon monitored usage data.

The utilization of the proxy pattern denotes the problems that are solved by a runtime persona: on the one hand, they obscure individual aspects of the real user, such as sensitive information like the `real name`; this is referred to as a *protection proxy* [117]. On the other hand, they can provide information about the real user that is available even during its absence; this is referred to as a *remote proxy*. In addition, as it may represent more than one real user, it has the ability to summarize similar characteristics across them and thereby augment information; we call this a *cluster proxy*. They keyword `derive` further highlights that the runtime persona is computed based on data that is produced by the real user.

While the users are using the `Application`, `PersonaKit` is observing the interaction and instantiates a `Session` that captures multiple `Characteristics`, such as the users' screen touches, views that are visited most often, different statistics about the time they spend during the interaction, as well as information about

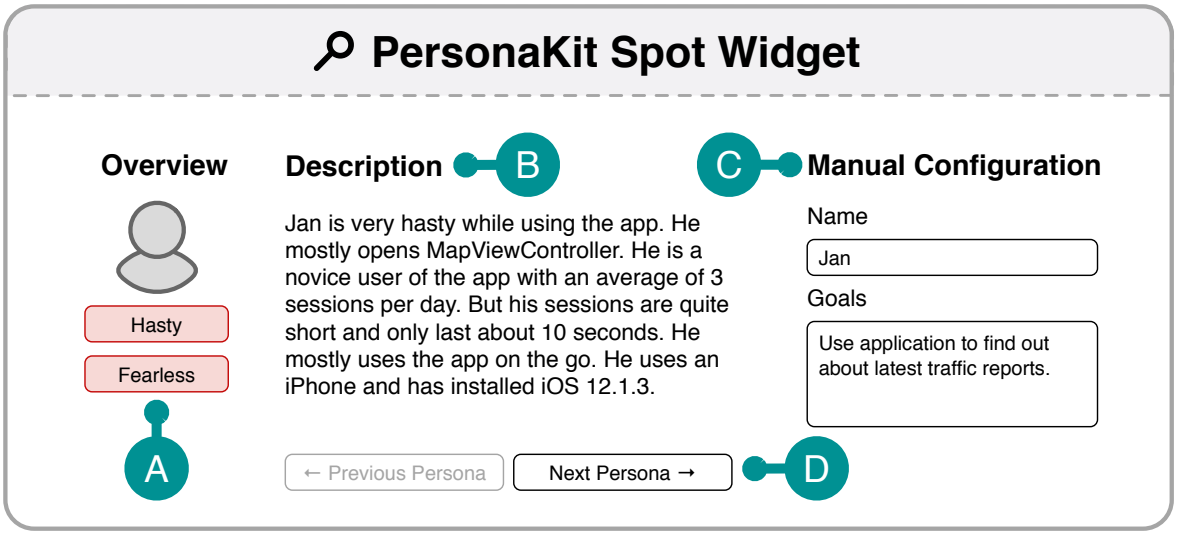


Figure 8.13: Sketch of a spot widget for PersonaKit usage knowledge.

the device, such as the type, the operating system version, or the font scale. After the user stopped interaction with the application, PersonaKit uses the session information to (re-)generate runtime personas.

The runtime personas are the result of a clustering procedure that tries to identify data points that share similarities, i. e., collections of data points which have a low distance in their respective distance matrix. In order to incorporate both continuous and categorial data types, PersonaKit relies on the combination of the Gower distance [123] and the *Partitioning Around* algorithm which allow the clustering of the mixed data types [164]. Following a final processing step, i. e., the specification of optimal numbers using the silhouette width as a validation metric [164], a set of runtime personas is provided by PersonaKit.

Each runtime persona consists of `Traits` that are compiled by one or more characteristics. For example, *hasty* denotes users that only briefly interact with an application and tend to cancel an operation if it takes too long. As another trait, *fearless* might describe a user that immediately closes alerts without having had the chance to properly read its message. Runtime personas further encompass textual `Descriptions` that provide a human-readable, but automatically generated text that summarizes characteristics and traits.

Usage Knowledge Spot Widget

Besides the ability to process usage data, PersonaKit also provides a visual representation of the usage knowledge: Figure 8.13 presents a spot widget for PersonaKit usage knowledge.

With respect to the example scenario that we described in Section 8.2.1.3, this widget can provide further insight to understand the needs and problems of the

user. The widget presents a lists of traits, i. e., hasty and fearless Ⓐ, that were detected by PersonaKit. This helps the developer with quickly understanding the core attributes of the users the runtime persona is reflecting. As mentioned, in contrast to traditional personas, runtime personas are built from monitored data. Therefore, the textual elaboration on the runtime persona that are presented in the widget as descriptions Ⓑ are driven by the application and its content. In the example, the description can help the user to understand that there is a user group that is considered a novice group, however, is mainly using the application in a mobile context. This raises the concern that—while they are likely to benefit from the new map layer alter feature—they are unlikely to discover the feature as they are unfamiliar with similar interaction concepts. Given the importance of goals for the traditional persona concept, the developer has the option to manually enrich the runtime persona for future use cases, or to share this information with fellow developers (Ⓒ). Likewise, they can add a name to the runtime persona. However, this shall be treated with cautions, as it should only serve as a reference point for the runtime persona. Finally, if PersonaKit discovered more than one runtime persona, the developer has the option to traverse the collection of them using the lower part of the widget (Ⓓ).

8.5.2 Related Treatments

The generation of personas has been in the focus of ongoing research in various domains. In the following, we list three areas of interest that relate to PersonaKit and outline similarities and differences.

First, Zhang *et al.* present a *quantitative bottom-up data-driven approach* to create personas [322]. They create personas primarily from click streams, i. e., telemetry data. Based on this data, they were able to derive 10 workflows and identified 5 personas, which they validated with domain experts. While they also rely on user behavior data, PersonaKit strives for a generic approach to support developers with personas that are created from a variety of data, such as the usage of involved features or their personal traits. In addition, PersonaKit encompasses a visual representation that includes a textual description of the derived runtime personas.

Second, several researchers have investigated the automatic generation of personas based on user interactions or social media data [162]. Other automatic generation approaches rely on additional data that is provided by external systems, such as social media platforms [14, 162]. They share the goal of PersonaKit to provide a means to derive personas and thereby insights on user groups without the need to involve laboratory experiments that require high time efforts. However, PersonaKit is focused on deriving such insights based on the interaction within applications.

Third, semi-automatic procedures are used to support the personas creation process to augment additional activities. Alvertis *et al.* describe the use of auto-generated personas from external sources, such as Facebook or YouTube, to provide a means

for software development teams to which they can relate requirements and respective documentation [13]. This can help to drive the overall development process, as it provides the context for collaboration between developers [13]. By using a semi-automatic approaches, Rahimi and Cleland-Huang propose to create and use personas to prioritize feature requests and facilitate other project-related activities, such as stakeholder communication [251]. PersonaKit shares the ability to allow developers to provide additional information to the generated personas, such as a persona name or goals, in order to augment it with rich information.

Chapter 9

The Control of Continuous User Understanding

> *"The fact that computers are increasingly being introduced into [various] environments does not imply that the human is being replaced. Instead, the human's role is changing. The human who used to be responsible for direct control of these systems is now becoming more of a monitor and supervisor of the computers that do the direct controlling [291]"*
>
> — WILLIAM B. ROUSE [262]

So far we focused on static aspects of the **Cuu**^{SE} framework that reflect the treatment to extract usage knowledge from usage data, making it accessible through visual components, allowing shared reference points, as well as integrating **Cuu**^{SE} with external systems. This chapter addresses the way developers make use of the treatment, i. e., how they utilize the **Cuu** workbench and **Cuu** kits in order to derive actionable insights for the implementation of a feature.

In Section 9.1, we motivate the need for an extended perspective on dynamic aspects when utilizing user feedback during agile software development processes such as CSE. This becomes in particular relevant as knowledge from the HCI community is combined with software engineering, i. e., agile development knowledge [61]. We introduce a workflow model that reflects foundations for a guideline toward the continuous user understanding activity demanded by practitioners in Chapter 5. In Section 9.2, we focus on one of the most relevant activities from the workflow, i. e., the analysis of usage knowledge by the developer. The description of the activity supports developers and satisfies the request of Moran, who is in search of *"[a] handbook for [..] a methodology [for user interface design], explaining the relevant kinds of experiments and leading one around the pitfalls"* [210]. We summarize the chapter with an analysis of related work in Section 9.3.

9.1 Workflow Design

Different terms are utilized to describe the work with user feedback, usability engineering is typically based on more heavyweight and traditional lifecycles. For instance, Mayhew's usability engineering lifecycle [205] differentiates between three main phases, which start with a focuses on requirements analysis and encompass a heavyweight phase of design, testing, and development techniques. Only during the so-called installation phase, user feedback is incorporated.

User-centered design is considered more seriously at large if a computer-assisted usability engineering platform is available [284]. Continuous experimentation focuses on the idea of build, measure, and learn cycles that are repeated several times by using a technical infrastructure [95]. However, existing publications in the area of continuous experimentation do not specify in detail how user feedback can be aggregated, analyzed, and turned into findings that improve the usability of the features under development [96].

Agile and CSE processes focus on small iterations which lead to product increments features at an early stage. However, permanent redesign of the user interface can lead to massive problems for the usability of an application [128]. There is a high effort to instantiate usability engineering in processes that focus on rapid development, which could hinder development teams. In general, existing agile and CSE processes do not specify how to apply usability engineering techniques.

To understand usability issues and to generate recommendations for improving a feature, traceability is indispensable. Cito *et al.* point out that diagnosing problems during run-time is in particular challenging when code is rapidly changed [62]. With respect to usability evaluation, mapping feedback to the right aspect of the feature under development is required. However, user feedback tends to be disconnected from the feature under development.

CSE provides the opportunity to collect a variety of user feedback that can be implicit, e. g., through user monitoring. The amount of data generated by monitoring takes time to process and may be difficult to understand. Barik *et al.* report that the combination of multiple data sources from usage logs remains a major challenge when making decisions [24]. This might discourage developers from deriving insights from the monitored user data; instead, they might focus only on explicit user feedback with a high information density. Opportunities for feedback analysis remain unused due to the high amount of recorded usage data.

In the following, we introduce the **Cuu** workflow that integrates usability evaluation into CSE. The motivation is to create an understanding how CSE can be combined with usability engineering aspects to create a hybrid, usability-driven, and feature-driven development lifecycle. The process model focuses on applications based on graphical user interfaces.

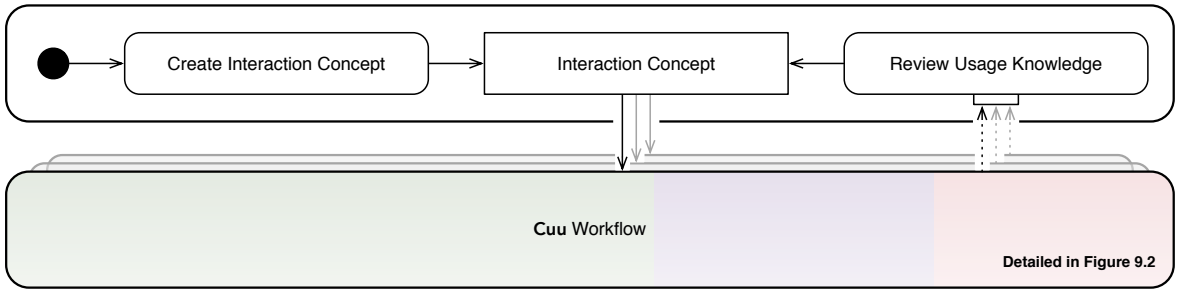


Figure 9.1: Related activities of the **Cuu** workflow (UML activity diagram).

9.1.1 Model

Several process models on how to organize and structure development efforts in such environments have been proposed, for example, SCRUM [282] or KANBAN [15] However, they face limitations, such as the ability to react to events within a certain scope, such as during a sprint in SCRUM. At the same time, they focus on management activities and remain purposely vague when it comes to details regarding the implementation. As **Cuu**$^{\text{SE}}$ framework is designed for being applied in an CSE environment, we decided to extend the RUGBY process model [176, 175]. As part of the CURES vision described in Section 3.2, we already described the core workflows of RUGBY. However, while RUGBY puts a special emphasize on the utilization of user feedback, it is focused on explicit user feedback and omits specifics of implicit user feedback that is derived from user monitoring, as well as does not discuss the need for tool support. To this end, the **Cuu** workflow provides new activities and roles to enable continuous user understanding in CSE. In Figure 9.1, we depict the context in which the **Cuu** workflow is embedded.

The **Cuu** workflow describes activities that focus on the improvement of a single feature. However, the development of multiple features can be parallelized—which is indicated by the multiple instances in Figure 9.1. The foundation for the workflow is the creation of a general *Interaction Concept*. It forms the baseline in terms of high-level usability requirements, such as regarding user profiles, general design principles, usability goals, platform capabilities constraints, or style guides as described by Mayhew [205]. The interaction concept encompasses the overall application structure in which features can be incorporated and thereby forms the starting point for future development. This is important because in particular nonfunctional requirements, such as usability, do not emerge from small increments [128]; they are embedded in a larger scope, which determines their manifestation.

The interaction concept is initially set up at the beginning of a project, i. e., *create [an] interaction concept* is performed once. It serves as the input parameter for an instance of a **Cuu** workflow, which, in turn, might also result in usage knowledge that can contribute to an improved interaction concept. As part of the *review usage*

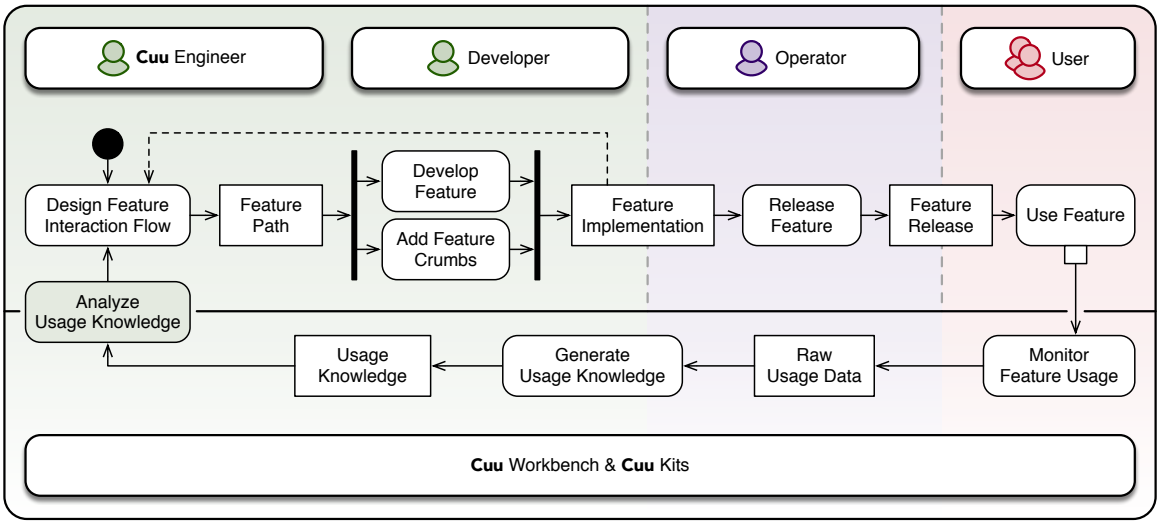


Figure 9.2: The **Cuu** workflow (UML activity diagram).

knowledge activity, a stakeholder can assess whether the derived insights during the performance of the workflow may be worth considering for future features. This generalization usually requires the transformation of usage knowledge into a usage pattern, as we described in Section 2.1.3.

We present the main **Cuu** workflow in Figure 9.2; it encompasses multiple roles, i. e., a ***Cuu*** *Engineer*, a *Developer*, an *Operator*, and the *User*. A **Cuu** engineer is concerned with the validation of requirements, i. e., ensuring that the feature under development fulfills the users' needs. On the other hand, the developer is in charge of verifying requirements, i. e., ensuring that the feature under development fulfills the specification described by the **Cuu** engineer. The role of the operator is described in more detail by Krusche *et al.* [176]: they use the term release manager. While its competence is wide-spread, with respect to the **Cuu** workbench, the role ensures that the application is delivered to users. One member of a development team fills in one or multiple roles. Most likely, the roles of a **Cuu** engineer and a developer are taken by the same individual. Note that practitioners also advised to consider fellow developers as users, as described in Section 5.1.2.3.

Based on the general interaction concept, the creation of every new feature starts with the activity *Design the Feature Interaction*. This activity may be initiated through an external feature request by a customer or when a new sprint is started and the implementation of a backlog item of the sprint backlog is imminent. The **Cuu** engineer describes the implementation of the feature request in terms of its user interface; they consider the flow needed to achieve the requested feature, which addresses involved views, buttons, and any other interaction element. The result is the feature's interaction flow, which can be represented in a formal artifact such as a scenario, which details events between user and application [56]. In the **Cuu** workflow, this is reflected in a *Feature Path*, which may be specified using the **Cuu** dashboard.

The feature path serves as the starting point for a developer, who is in charge of the realization of the feature specification as part of the activity *Develop Feature*. At the same time, it is their responsibility to *Add Feature Crumbs* to the source code. The feature crumbs, which are used to describe a characteristic of a feature as described in Section 7.2, are added as code-level statements, for example in the event handler of a button or during the transition from one to another view. The result of both activities is compiled by the developer to a new *Feature Implementation*. Findings during the development might influence the feature interaction flow and can lead to a new feature path version. We acknowledge this loop in the workflow using a dashed line in Figure 9.2.

Through an automated setup, the feature increment is handed over to the operator, who is in charge of *Releas[ing the] Feature*. This activity makes sure that a *Feature Release*—a release that includes the feature under development—is delivered to the right audience of users. As described, the workflow omits details on the release process that are not relevant when using **Cuu**$^{\text{SE}}$; they are detailed in the RUGBY process model by Krusche *et al.* [176].

Users receive the release and thereby access to the full application that—among others—includes the feature under development; they start to *Use [the] Feature*. While doing so, they produce a variety of *Raw Usage Data*, such as interaction traces. During the *Monitor Feature Usage* activity, both the **Cuu** workbench and the **Cuu** kits collect the raw usage data. They also *Generate Usage Knowledge* based on the usage data and make the result, the *Usage Knowledge*, accessible to both **Cuu** engineer and developer through the **Cuu** dashboard. In case the generated *Usage Knowledge* are of interest for other features, they can influence the overall *Interaction Concept*, as described with Figure 9.1. The **Cuu** engineer, or in collaboration with the developer, *Analyze[s the] Usage Knowledge* with the goal to improve the current feature interaction flow. We detail the analysis of usage knowledge in the following sections.

9.1.2 Instantiations

Different variations of how the activities of the **Cuu** workflow can be instantiated based on Figure 9.2 help to illustrate the utilization of the workflow. Figure 9.3 shows an informal timing diagram with four exemplary instantiations of **Cuu** engineers' and developers' activities (①, ③, ⑤, ⑦,), and users' ways of providing feedback (②, ④, ⑥) as lifelines. The operator facilitates between both other roles; we use the abbreviation *FR* to indicate when a new version has been bundled and released to the user, while its main purpose is to validate the feature under development. Underneath—summarized by *Monitor Feature Usage*—multiple activities represent different ways of providing raw usage data by users which is collected and monitored by kits. This is a non-exhaustive list taken from Chapter 8, any *Kit* can be considered. Their result is bundled as a set of usage knowledge *UK*.

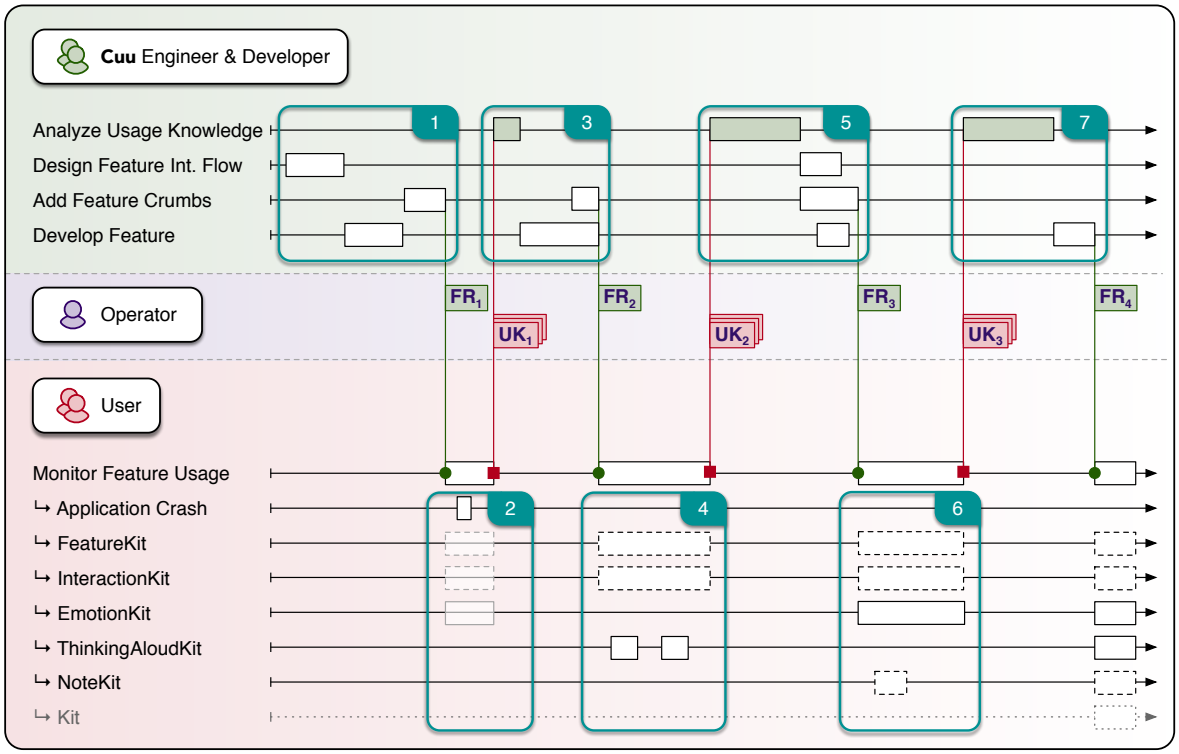


Figure 9.3: **Cuu** workflow instantiations (informal UML timing diagram).

We rely on the *Maps Layer Alter* feature to describe the instantiation of activities; the four upper groups in Figure 9.3 are reflected in the four commits in Figure 7.15, while $FR_{1,2}$, FR_3, and FR_4 represent version 1, 2, and 3, respectively, in Figure 1.1. In Figure 9.3, we omitted the **Cuu** workbench which supports the developers during most of the activities described in the following.

Initially, the first version of the map layer change is implemented ①. A **Cuu** engineer starts to design a feature interaction flow. Based on this specification, the development of the feature follows; hereafter, developers start adding feature crumbs to the new code additions. The operator creates a release which can be used by the user. Note that ① depicts a sequential procedure: Besides the addition of the feature crumbs, this is similar to traditional development and usability engineering.

Almost instantly after the release ②, when users start using the feature increment and reach the latest source code addition, it becomes obvious that there is a code-related bug that causes the application to crash.

While raw usage data is collected, it is not further considered when the *UK* is send to the **Cuu** engineer for analysis ③. The analysis is straightforward, the reason for the crash can be identified instantly using the *UK*; only minor changes need to be performed regarding the arrangement of feature crumbs; most of the improvement is achieved by code adaptions. A new release FR_2 is pushed to users.

This version includes the map layer alter feature that is located within the side

menu; the users start using it and feature usage is monitored ④. This time, the crash is resolved and raw usage data for the full feature path can be monitored. While the user is using the application, multiple feature crumbs are triggered and recorded by *FeatureKit*. In addition, interaction data, such as tap and view events, are monitored. Both are represented as dashed lines, as they are recorded as discrete values and represent event-based information. The users do not provide access to facial expressions, maybe because their device does not support its recording or because they declined usage hereof; we discuss this in Section 9.2.1.2— in any case, no continuous stream of emotions can be monitored. However, two thinking aloud sessions were performed by users over a period of time, which represent a continuous stream of usage data. All *UK* is send back to the **Cuu** engineer and developer.

The *UK* analysis time increased ⑤, because it requires effort to understand and combine the usability findings. We detail this process in Section 9.2. Overall, while the insights from FeatureKit and InteractionKit indicate that the user can make use of the map alter feature, in the two thinking aloud sessions the users elaborate on their needs for such as feature during other activities, when the side menu is not in reach. This motivates the **Cuu** engineer to introduce version 2, i. e., the rounded-circled button as seen in Figure 1.1. After the decision has been made, the three consecutive activities, i. e., the design of the feature interaction flow, the addition of feature crumbs, and the development of the feature improvement, are performed simultaneously.

Again, feature, interaction, and emotional usage data are collected ⑥. One user provides written feedback using NoteKit, which encompasses qualitative feedback regarding the feature increment. No thinking aloud sessions were triggered.

Based on the UK_3, the **Cuu** engineer analyzes the usage knowledge ⑦. In particular provided the emotional responses, the users appear to be confused about the feature change. This is supported by the interaction data and the feature path observations: The successful completion of features may have decreased. As a result of the analysis, the **Cuu** engineer decides to introduce the tooltip as shown in version 3 in Figure 1.1. Only the code is adjusted based on the findings; no feature crumbs are added, nor the design of the feature is changed. This leads to an interesting situation: both releases FR_3 and FR_4 can be compared in terms of performance using the compare widget as described in Section 7.3.2.

9.2 Analysis of Usage Knowledge

The most important activity during the **Cuu** workflow is *Analyze Usage Knowledge*, which is emphasized its green color in Figure 9.3. During this activity, the **Cuu** engineer combines multiple usage knowledge types to identify the need to improve a feature or address usability issues. This may lead to a new feature path version.

9.2.1 Conceptual Aspects

The analysis demands high cognitive capabilities from the **Cuu** engineer. To address this challenge, the *Raw Usage Data* needs to be available in form of *Usage Knowledge*, as shown in Figure 9.2. In the following, with respect to the usage knowledge that is provided as an input parameter to the analysis activity, we distinguish between levels of automatization in processing the raw usage data, which leads to dependencies between usage data, as well as the ability to filter usage knowledge using criteria during the **Cuu** workflow.

Automation of Usage Knowledge Generation and Capture

As part of the activity *Generate Usage Knowledge*, raw usage data is aggregated and processed by **Cuu** kits. From the perspective of automation, two aspects are relevant for the **Cuu** engineer.

On the one hand, the **Cuu** kits' expertise to automate the transformation of usage data affects their results. As indicated in Figure 7.2, user feedback—and in particular the usage knowledge—can be of different granularity levels. For example, the automation of a knowledge source may be simply to remove nosy data or any other distraction that is not relevant for the usage data. In this case, the engineer is confronted with more efforts to derive meaningful insights from the user feedback, while—in contrast—the need for action of usage knowledge of level 2 or higher (Figure 7.2) becomes more evident with every level and thereby reduces the engineers engagement.

On the other hand, some knowledge sources require the activation by the **Cuu** engineer in order to collect a continuous input stream. For example, usage knowledge from ThinkingAloudKit (Section 8.3.1), i. e., the verbal description of a user that is automatically analyzed, transcribed, and classified using a machine learning classifier, requires the manual activation by the developer, as the process of collecting the usage data interrupts the users interaction flow.

Overall, the *Analyze Usage Knowledge* activity benefits from a high level of automation in order to provide structured, preselected, and processed raw usage data for easy further use by the **Cuu** engineer.

Dependencies between Usage Knowledge Types

Some usage data or its processed results can act as the source for usage knowledge on its own, others provide support for other knowledge sources. We rely on feature crumb observations, i. e., the number of times a particular crumb was triggered and the processed state, as a main input parameter for other knowledge sources. Likewise, interaction data is considered a low-level usage data, as it needs to be collected to empower other knowledge sources.

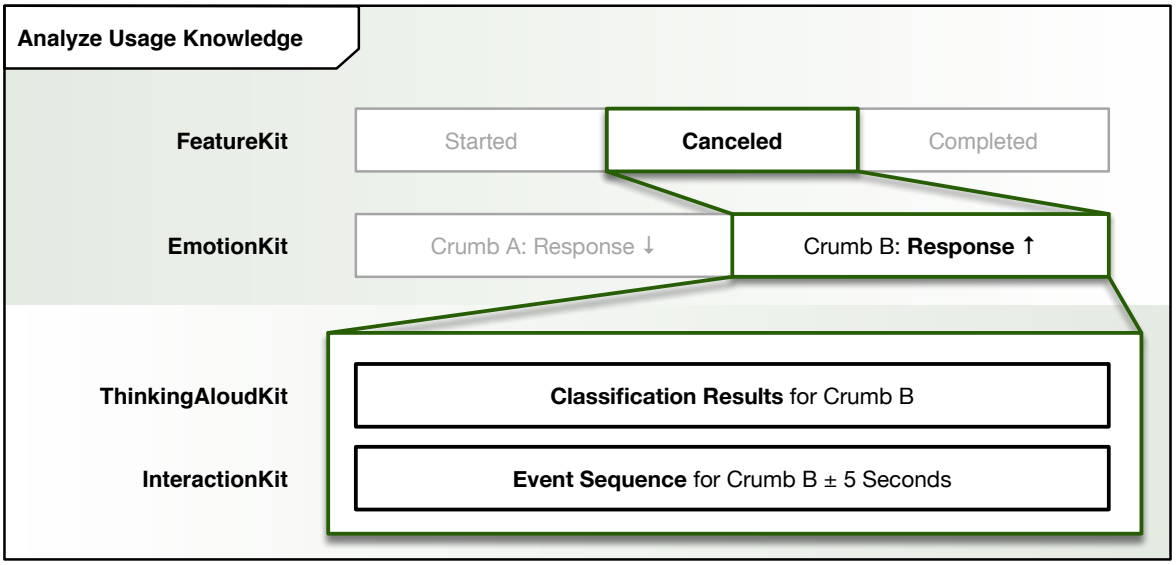


Figure 9.4: Filtering usage knowledge during the *Analyze Usage Knowledge* activity.

For **Cuu** kits, this has two effects: First, a knowledge source that applies machine learning can rely on InteractionKit data for both their *process* and *fuse* methods (Figure 7.2). Second, feature crumbs form a reference point for usage information; knowledge sources can rely on FeatureKit for their *relateTo* method. As a result, we refer to usage data monitored by InteractionKit and FeatureKit as *independent* usage data, as they do not require other knowledge sources. This enforces the need to monitor such data at any time, as described in Section 8.1.2.

We refer to **Cuu** kits such as EmotionKit or BehaviorKit as *dependent* usage knowledge sources, as they rely on usage data from other kits. In addition, their dependency is further emphasized by the fact that users might opt out of monitoring and collecting respective usage data. Also, such usage data might not be available because a user might be concerned about privacy. Another reason may be the lack of a specific sensor that is required, e. g., a 3D camera for measuring facial expressions which is currently only available in high-end consumer devices.

Filtering of Usage Knowledge

We indicated in Section 9.1 that opportunities for the evaluation of usability as well as improvement of features are left unused given the high volumes of usage data. Therefore, the **Cuu** workflow encourages filter criteria for usage knowledge. We demonstrate the reduction of relevant usage knowledge during the *Analyze Usability Findings* activity in Figure 9.4.

During *Generate Usage Knowledge*, FeatureKit assesses the observed feature crumbs and compiles all user sessions by their state *Started*, *Canceled*, and *Completed*. As canceled features observations are likely to contain issues, respective sessions should be

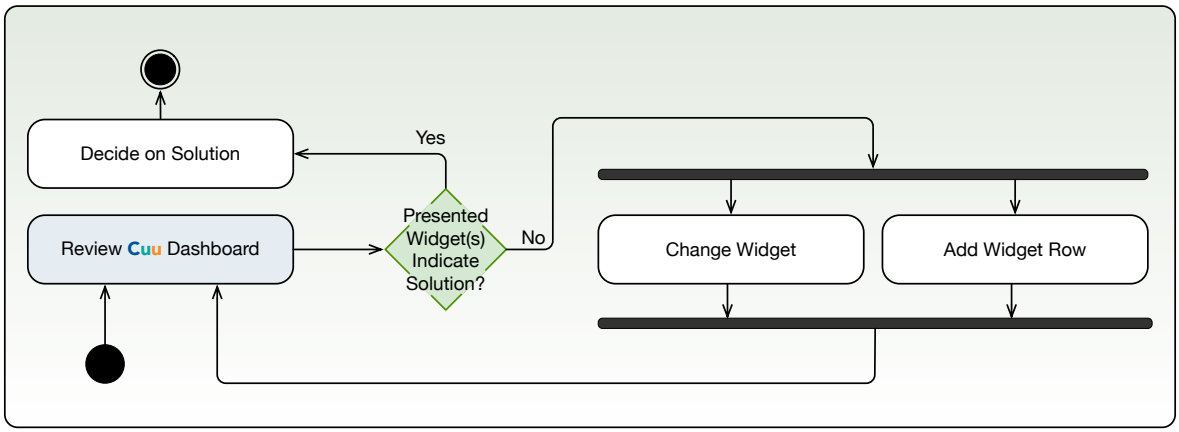


Figure 9.5: **Cuu** dashboard interactions during *Analyze Usage Knowledge* activity (UML activity diagram).

investigated in more detail. Using the emotional responses provided by EmotionKit, crumbs within this feature that show a high emotional response can be selected as candidates for a more detailed investigation. Based on this selection, other usage data can be incorporated and used in parallel to help finding the problem: classified thinking aloud protocols by ThinkingAloudKit help to identify the actual problem, while event sequences from InteractionKit provide additional context.

9.2.2 Utilization of Workbench

The **Cuu** workbench including the **Cuu** kits as depicted in Figure 9.2 enable the effective implementation of the workflow. The tool support is required to automate the *Monitor Feature Usage* activity and is indispensable to enable continuous user understanding with respect to changing feature increments, as described in Chapter 5.

In the following, we describe how the *Analyze Usage Knowledge* activity is realized within the **Cuu** dashboard. This is an iterative and non-deterministic process, as depicted in Figure 9.5. Bruegge and Dutoit use the metaphor of an archaeologist to describe the incremental creation and development of a model [45].[30] The same metaphor can be applied in creating a model of the user and their needs.

The **Cuu** engineer starts by *Review[ing] the **Cuu** Dashboard*. They are presented with a single widget that visualizes usage knowledge that was produced based on a kit's expertise. The widget might already *[...] Indicate [a] Solution*. In this case, the engineer can immediately *Decide on [a] Solution* and leave the analysis activity in order to continue with *Design [a] Feature Interaction Flow* (Figure 9.2).

[30] *"Fossil biologists unearth a few bones and teeth preserved from some dinosaur that no one has ever seen. From the [..] fragments, they reconstruct a model of the animal, following rules of anatomy. The more bones they find, the clearer their idea of how the pieces fit together and the higher the confidence that their model matches the original dinosaur. If they find a sufficient number of bones [and] teeth [...] they can almost be sure that their model reflects reality accurately, and they can guess the missing parts."* Bruegge & Dutoit [45].

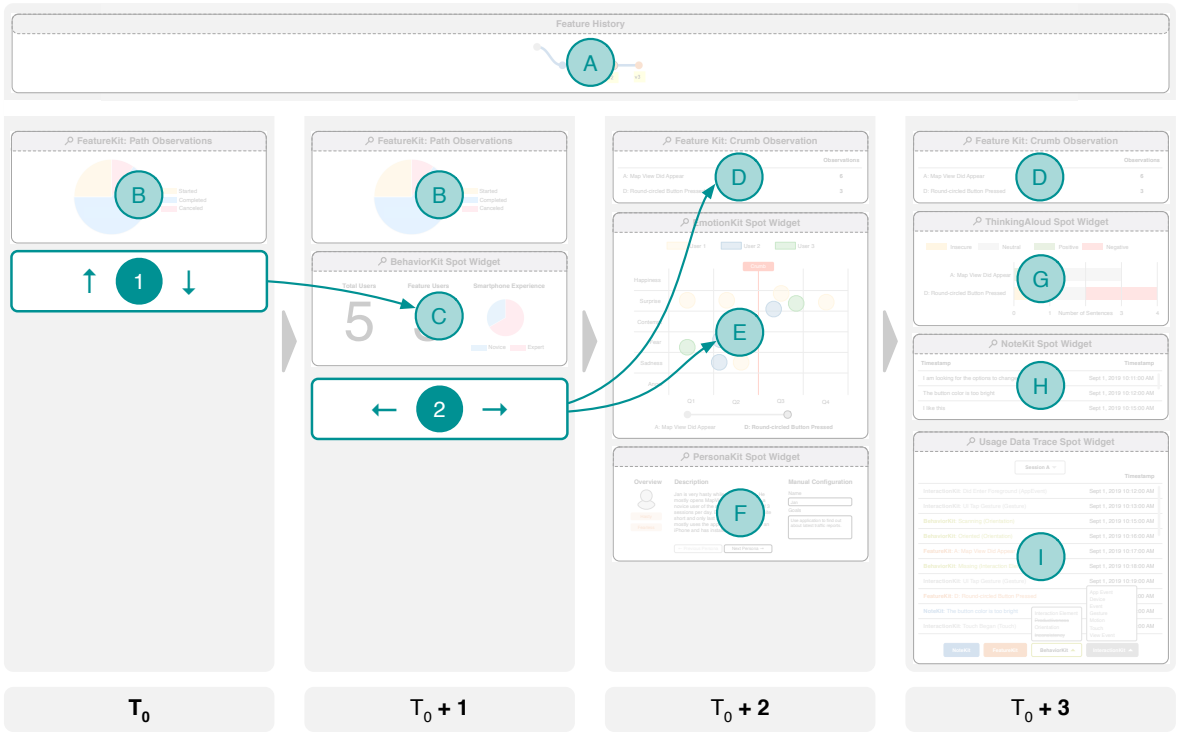


Figure 9.6: Change of the **Cuu** dashboard configurations over time.

Other cases might require an additional or multiple widgets in order to find a solution. Then, the engineer needs to modify the configuration of widgets that are presented in the **Cuu** dashboard. They can either *Change [a] Widget* or *Add [a new] Widget Row* that hosts a new set of widgets. This enables the simultaneous visualization of usage knowledge from multiple kits at the same time. Hereafter, the engineers restarts the review of the **Cuu** dashboard.

We exemplify the engineers' work with the dashboard in Figure 9.6. Its upper part, marked with Ⓐ, represents the springboard; this remains the same for the four configurations that are depicted below as vertical columns.

Figure 9.6 outlines how the engineer would *Analyze Usage Knowledge* for version 2 of the map layer alter feature (see Figure 1.1). Every column describes a configuration of one or more widgets that evolved over time. The widgets presented in Figure 9.6 depict blurred versions of the widgets introduced in Chapter 8 to increase the readability—a non-blurred version is presented in Appendix C.3.

The developer starts off the analysis by inspecting a first widget. The observed feature paths provide a good entry point (Ⓑ), as they instantly outline the performance of the feature. As it appears that there are three canceled feature path observations, the **Cuu** engineer starts adding a widget; while at least one widget must be visible at any time, new widgets or additionally presented widgets are stacked vertically ①. The dashboard provides a dedicated interface for this purpose as depicted as Ⓜ in Figure 7.15.

The engineer decides to add a BehaviorKit widget (Ⓒ), and thereby creates a new configuration of the dashboard at T_0+*1*. With the help of this second widget, it becomes clear that there are three users struggling with the successful completion of the feature; and that in particular novice users are affected. Now, the engineer is sure that there is a problem for which they need to provide a solution and start further investigation. Following the pattern described by ②, they can select another widget by swiping to the left or the right of the widget; they can also use the dot area for this purpose as described by Ⓛ in Figure 7.15.

With T_0+*2*, they change the first two widgets to a feature crumb observation Ⓓ and an EmotionKit widget Ⓔ, respectively. They allow the identification of the feature specific that appears to cause a problem: the crumb *D: Round-circled Button Pressed* is only barely triggered; while the users who do make use of it show a high emotional responds toward it. The addition of the PersonaKit widget Ⓕ provides further context to the user audience.

Now that the **Cuu** engineer identified the occurrence of a problem, they make use of explicit user feedback: in T_0+*3*, they replace parts of the previous configuration and add a ThinkingAloudKit Ⓖ and NoteKit widget Ⓗ. They provide the reason for the users' confusion. The usage data trace view in Ⓘ supports the developer in understanding user feedback that does not provide a reference to the feature under development.

This workflow allows to adopt usability engineering techniques to CSE. The developers maintain a relationship to the feature under development when assessing the user feedback. Notice that the workflow is meant as a process guideline, rather than a strict procedure the **Cuu** engineer needs to adhere to. The flexible structure of the dashboard should encourage the **Cuu** engineer to explore different usage knowledge and their combination—maybe even in a collaborative environment with fellow colleagues and developers. The exploration aspect is further supported by technical implementation details, such as support for physical swipe gestures when the dashboard is viewed on a tablet, or support for adding and removing widget rows using shortcuts in case a keyboard is available. These additions should increase the usability of the dashboard and thereby facilitate the analysis of usage knowledge. Overall, by combining the usage knowledge from two or more widgets, the **Cuu** engineer becomes a knowledge source on their own, as they also perform a *fuse* method on the way to extract tacit knowledge from the widgets.

9.2.3 Discussion

The **Cuu** workflow reflects a central artifact of the **Cuu**$^{\text{SE}}$ framework, as it describes the glue between the **Cuu** workbench, i. e., the dashboard, as well as the usage knowledge provided by the **Cuu** kits and the feature crumbs concept. In the following, we discuss three aspects which we expect relevant in this context.

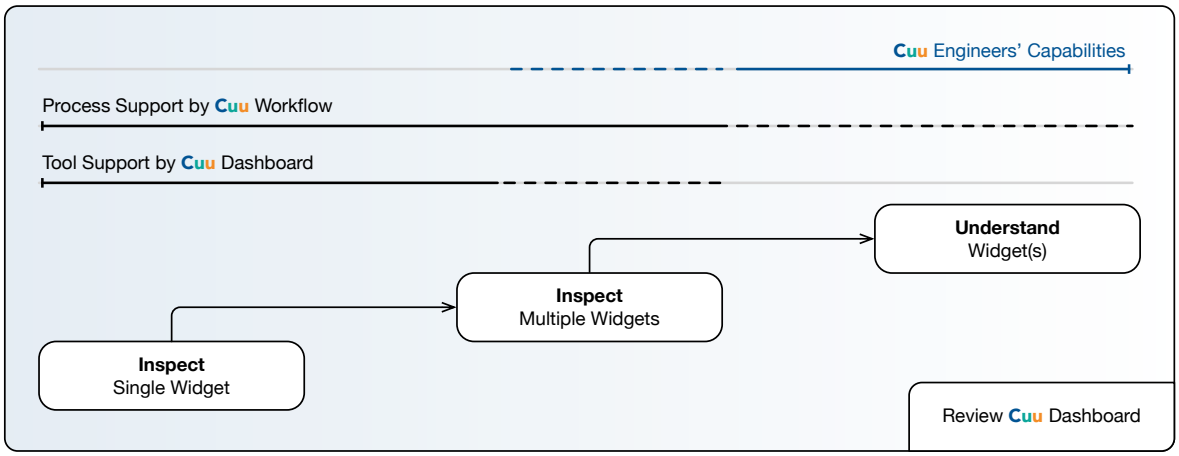


Figure 9.7: **Cuu** engineers' interactions during *Review **Cuu** Dashboard* activity.

Dependency on Activities

A successful implementation of the workflow model, and thereby the degree of the understanding of the user, depends on a few activities that are performed by developers. Most notably, by *Add[ing] Feature Crumbs* to the source code as described in Figure 9.2, the developer lays the foundations for deriving insightful usage knowledge. This can be illustrated when comparing feature crumbs with break points for debugging: Stopping the execution of a program at the *right* time might reveal the *right* information for improvement. In contrast, when developers define incorrect or misleading combinations of feature crumbs, the usefulness of derived usage knowledge is limited. As a consequence, adding feature crumbs to code embodies an important activity, since it directly influences the quality of the outcome of the workflow.

Demand on Engineers

The workflow model in Figure 9.1 suggests a flow on how development teams can integrate a continuous user understanding activity into CSE. However, it requires new capabilities during the activity *Analyze Usability Findings* in understanding and combining the usage knowledge. We already discussed this challenge by outlining the implications of a usage monitoring concept for software engineering processes in Section 7.2.2.2. With Figure 9.7, we summarize how tool and process support precede, but do not replace, the **Cuu** engineers' capabilities; the figure depicts the engineer's considerations during the *Review **Cuu** Dashboard* activity in Figure 9.5.

This is a non-trivial activity; the complexity of usage knowledge for inspection increases with every addition of a widget and peaks in the understanding of such by the **Cuu** engineer. We argue that by relying on the **Cuu** dashboard and the **Cuu** workflow, the overall complexity in understanding user feedback is reduced.

The complexity needs to be mediated through optimizations in the *Generate Usage Knowledge* activity and improvements to the workflow. We suggest to investigate on this as part of future work.

Measurement of Progress

The workflow model does not provide a means to measure the progress of improving toward the usability of a feature under development or understanding the users' needs. This aspect needs further consideration, as we expect that the utilization of the workflow by developers requires patience, since gainful insights only arise over time. However, following the principle of CSE in continuously releasing a new feature increment, we expect the quality to eventually converge to an optimum. The feedback of users can be understood as anecdotical evidence that the usability improved and the model of the user evolved—or requires more work, which results in new feature releases.

9.3 Related Workflows

Rosson and Carroll present an approach that allows to iteratively evolve a scenario and align the improvements with its implementation in code [261]. Therefore, they provide tool support in form of the *ScenarioBrowser*. Both the goal of iteratively improving a feature which is supported by tool support can be found in our work, namely in form of the **Cuu** workflow and the **Cuu** workbench. Rosson and Carroll further highlight the support of *"the analysis and documentation of design rationale for each scenario"* [261] with respect to the current development. The **Cuu** workbench supports the addition of other, non-usage knowledge related knowledge sources. For instance, we briefly outlined the ability to show a rationale graph as an example for decision knowledge widgets in Section 7.3.2. However, the **Cuu** workflow must be extended to incorporate related activities to reflect additional knowledge sources and to handle rationale discussion, extraction, and capture.

Paelke and Nebig investigate on the integration of agile methods and with usability engineering methods by incorporating mixed-reality development techniques [234]. They focus on applications that make use of generated computer graphics that are combined with real-world elements and strive to support their development process, as they identified multiple impediments for its successful implementation. Therefore, they introduce an extended process model. The **Cuu** workflow is not limited to the application that is being developed. However, it shares the need to introduce new roles: Paelke and Nebig described new roles, such as usability engineers and user interface designers, as well as mixed reality experts, and real users [234]. While the **Cuu** workflow builds upon the roles introduced by RUGBY, we also introduce an engineering role, the **Cuu** engineer, that focuses on the aspects

of user understanding. Following their model, we see future improvements for the **Cuu** workflow in adding an expert role that is knowledgable about about the technologies that are employed by the application under development; for now, we assume that this competence is covered by the developer.

Dzvonyar *et al.* introduce the *CAFE* system that consists of a framework and a process model for integrating context-aware user feedback into continuous software evolution [88]. They strive to extend the availability of context information to enrich the feedback provided by users which usually lack of motivation to provide extensive information, which is required by the developers to effectively address the user feedback during their development activities. With respect to the process, the Dzvonyar *et al.* extend the RUGBY process model, which we also used as the basis for the **Cuu** workflow. With respect to the framework implementation of CAFE, the authors focus on the collection of explicit user feedback, which is enriched by context-information. While such information could be reflected by kits, the **Cuu** workflow emphasizes the use of implicit usage information for knowledge gain. In addition, Dzvonyar *et al.* focus on how to turn feature requests, bug reports, and design requests into work items in and issue trackers, by which they achieve a *feedback traceability*. In contrast, the **Cuu** workflow establishes a *usage traceability*, which allows for a more fine-grained mapping of feedback to a feature increment.

Flaounas and Friedman extend the combination of agile practices and usability-focused work with a focus on business-related metrics [110]. They identify a gap between high-level performance indicators, e. g., the measurement of active users or revenue rates, and feature-focused development that typically involves minor improvements with respect to usability. To approach this problem, they described three nested cycles, i. e., the agile, tactical, and strategic cycle, while the tactical cycle encapsulates a mediating role that facilitates between short-term changes from the agile cycle and lagging metrics of the strategic cycle. The **Cuu** workflow focuses on the achievement of development-related activities that address the understanding of users and omits business-related metrics that address long-term effects of the newly introduced changes. However, future versions could implement the addition of a tactical and strategic cycle as described by Flaounas and Friedman [110].

As introduced in Section 2.1.4.2, the combination of CSE and usability engineering gains in relevance with respect to continuous experimentation. For example, Fagerholm *et al.* present ideas for an experimentation system to continuously perform customer experiments [95]: Therefore, they identify four main requirements, i. e., a system to release the feature under development that can be used by users, the control of experiment plans, the traceability of experiment results, and the ability to make adaptions to the overall business strategy. They describe an infrastructure that relies on the build-measure-learn paradigm. As continuous experimentation is concerned with formally addressing hypothesis that are defined beforehand for test-

ing, the infrastructure highlights the utilization of artifacts such as the experiment plans and experiment results next to the raw data that was collected. While the infrastructure shares similarities with the **Cuu** workbench, Fagerholm *et al.* describe the integration of development-related activities, such as usability engineering, as a cross-functional role which is aligned to the experimentation process. In addition, they do not specify on how the findings are generated from the collected results, as well as the content and sources of the raw data—which is a main focus of the **Cuu** workflow.

We conclude that, while continuous experimentation and continuous user understanding share similarities in tools and activities, they differ in their underlying goal: a quantitative against a qualitative understanding of user needs. At the same time, we argue that the **Cuu** workflow, and in general the **Cuu**$^{\textbf{SE}}$ framework, can be extended with additional components from continuous experimentation research: This could be either in form of new **Cuu** kits or more large-scaled extensions to the framework. The latter proposal could be of use when **Cuu**$^{\textbf{SE}}$ is scaled up for utilization with larger audiences of users.

Part IV
Treatment Validation

TREATMENT VALIDATION is concerned with the development of a theory that allows us to predict how a treatment will behave in a problem context [310]. This distinguishes it from the two last phases of the engineering cycle, i. e., the implementation and evaluation. In this part, we present three validation cycles.

First, we report on the validation of EmotionKit using a laboratory experiment, for which we studied the feasibility and applicability to derive emotions from facial expressions to detect usability problems. The qualitative and quantitative analysis of results can serve as a blueprint for the validation of other **Cuu** kits.

Second, we introduce the design of the **Cuu** syllabus as a means to disseminate **Cuu**$^{\textbf{SE}}$. We provide results from a quasi-experiment, in which we validated its applicability with a focus on how it affects organizational aspects, whether it supports developers in creating feature representations, and if it fosters the usage of **Cuu**$^{\textbf{SE}}$.

Third, we present results from a survey with student developers in which we strived for a comprehensive validation of the combined application of multiple artifacts of the **Cuu**$^{\textbf{SE}}$ framework. We describe the results of their perceived usefulness, their perceived ease-of-use, as well as their intention to use **Cuu**$^{\textbf{SE}}$ in future projects.

Chapter 10

Validation of Knowledge Sources

> *"The field [of user psychology] also needs to work more at bringing existing areas of cognitive psychology and traditional human factors to bear on user behavior. Some of the highly developed areas of psychology, such as perception, attention, and psycholinguistics, would seem to have quite a bit to say about user behavior, but a good deal of translation and tailoring may be necessary. The human factors work should be more straightforward to apply to computer hardware devices."*
>
> — THOMAS P. MORAN [210]

Following the development of treatment in Part III, we strive for the validation of the created artifacts. As described in Figure 1.3, we dedicated an empirical cycle to the validation of Knowledge Question 4. We want to put the focus on emotions as a representative study for validation. The procedure can act as a blueprint for the validation of other kits that includes the creation of RQs.

We introduced EmotionKit as a framework to efficiently detect emotional responses from users based on facial expressions without the need for machine learning classifiers in Section 8.2. In Section 10.1, we describe the extensive validation of the performance of EmotionKit. Therefore, we focus on the detection of usability problems as a means of user problems that occur during application usage. We explored the applicability and reliability of EmotionKit in a user study with 12 participants to show how subjects react to usability problems in a sample mobile application. We describe the results of the study as part of a qualitative and quantitative analysis.

In Section 10.2, we provide suggestions on how to validate other knowledge sources, such as ThinkingAloudKit, BehaviorKit, and PersonaKit, while presenting initial explorative results. We also provide an interrelated analysis of the kits to address the aspect that one kit might be better suitable than another for certain purposes. This provides a preliminary look on major characteristics which could provide a starting point for future assessment.

10.1 Validation of EmotionKit

The overall goal of the validation of EmotionKit is to show that facial expressions that were recorded using consumer hardware reflect an additional implicit knowledge sources for development in CSE environments. Therefore, we performed a study with 12 participants to evaluated EmotionKit's performance within a sample application that was seeded with usability problems.

In Section 10.1.1, we start by providing insights on the experiment design by refining the knowledge question into research questions and providing details on the research method. In a qualitative analysis in Section 10.1.2, we provide evidence that the framework is applicable for detecting user emotions from facial expressions. As part of a quantitative analysis in Section 10.1.3, we show that emotional responses can be detected in three out of four cases and that they relate to usability problems. We provide more insights on the validity of EmotionKit by demonstrating that phases of emotional responses can be derived on the monitored data, as described in Section 10.1.3.3. We summarize the validation with a discussion on the results in Section 10.1.4.

10.1.1 Experiment Design

The design of the experiment is driven by two aspects: first, we refine the knowledge question into four research questions. Second, we outline the research method.

Refinement of Knowledge Question

With Knowledge Question 4, we set out to investigate how a knowledge source performs as a means to collect usage knowledge. This is a critical aspect, as it directly influences both the effectiveness and the efficiency with which developers can extract tacit knowledge based on the visualization of the usage knowledge in widgets.

With respect to EmotionKit, we strive to validate two aspects: the applicability of the data collection and processing and the reliability of the usage knowledge. This reflected a multi-step research process, in which the research questions evolved over time. We started with the initial and explorative goal of verifying the applicability of EmotionKit. We summarize this as follows:

> **Research Question 13:** Can user emotions be derived from facial expressions using consumer hardware?

In a continuation of the efforts of RQ 13, we were interested in whether there exist certain emotion patterns that can point developers to a particular usability problem. We summarize this as follows:

Research Question 14: Do users show similar emotion patterns regarding usability problems?

Based on the insights from RQ 13 and RQ 14, we focused on the emotional responses of users, as they promised to be a suitable means to provide reliable usage knowledge for usability problem identification. We summarize this as follows:

Research Question 15: Does the emotional response of users reveal the existence of usability problems?

Finally, we were interested in increasing our understanding of the users' emotional response; the more insights on its structure are available, the better developers can interpret the results. We summarize this as follows:

Research Question 16: What is the structure of the users' emotional response with respect to a usability problem?

Research Process

We performed a laboratory experiment as introduced in Section 2.2.2.3 and describe its conduction in the following. This includes the description of the user study as well as threats to the validity of the study.

User Study This section describes the study approach to validate the applicability and reliability of EmotionKit to investigate on the possibility to track down usability problems based on emotions that were derived from facial expressions. Given that we implemented EmotionKit as an Apple iOS-based prototype as introduced in Section 8.2.1, we chose an Apple iPhone X and its camera system as the consumer test system that is available to users in everyday life.

Sample Application We created an application that displays six static views as sketched in Figure 10.1. The navigation between each of them includes a usability problem that we derived on the basis of the so-called *usability heuristics* [216].

In Figure 10.1, Ⓐ indicates the approximate location of the consumer device's *TrueDepth* camera. The sample application consists of eight views. For the static content of the sample application, we chose a topic that—to our understanding—does not trigger any particular emotion when consumed by a participant: the history of the Munich subway system. Each subway line (i. e., U1, U2, U3, U4, U5, and U6) is represented with one static view that displays textual informational about its past, such as the date of opening or specifics about its stops.[31]

[31] We extracted and adapted text from https://en.wikipedia.org/wiki/Munich_U-Bahn.

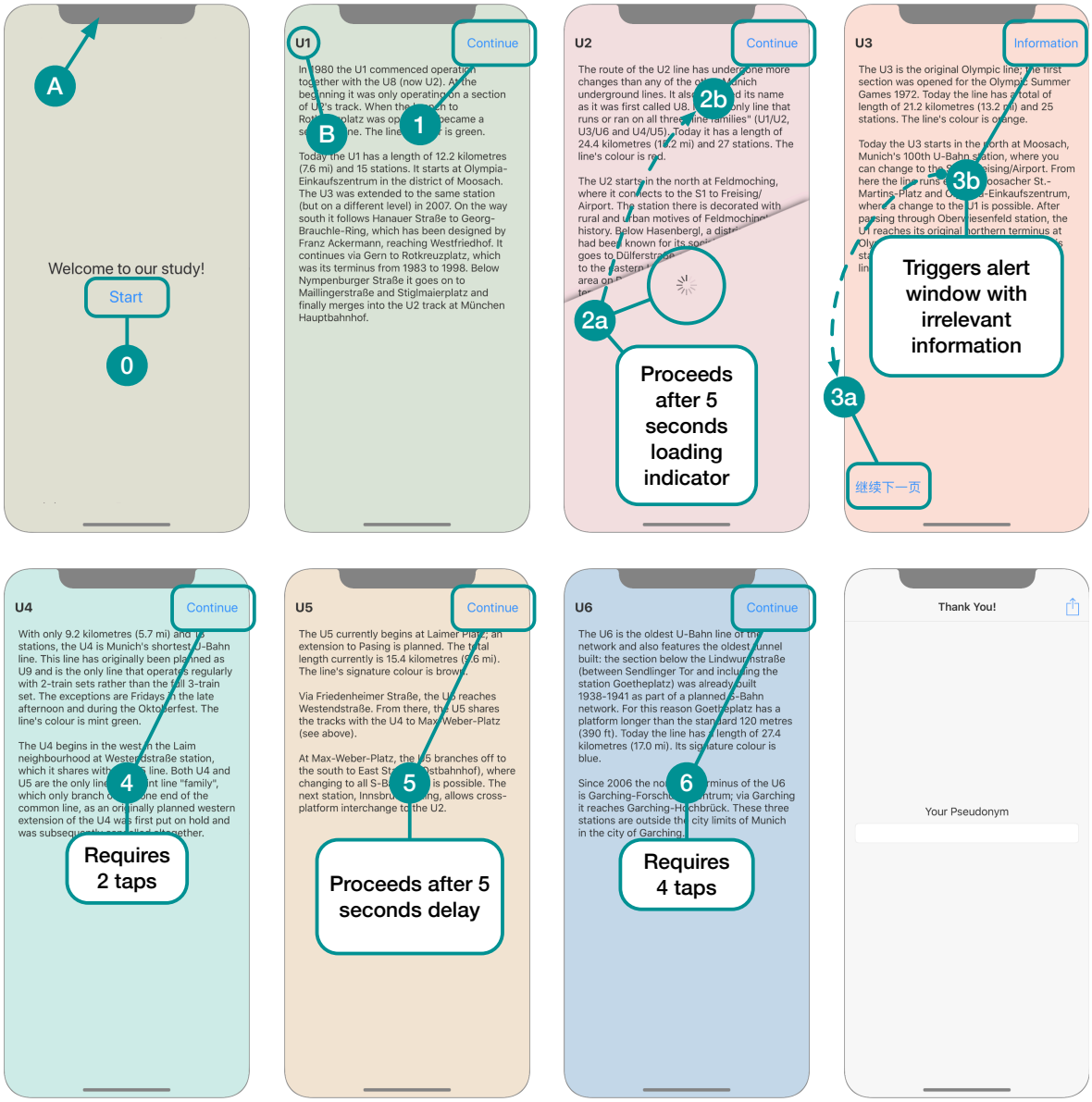


Figure 10.1: Sample application, adapted from Johanssen *et al.* [151] (© 2019 IEEE).

As part of the introduction phase of the study sessions, we asked each participant *Are you familiar with the U-Bahn?* to understand whether they are knowledgeable about the topic and to clear out any potential emotional constraints toward this topic which could affect the measurements.

For the application's dynamic interactive part, a *continue* button in the top right corner of the sample application, such as the one depicted by ①, serves the purpose of navigating to the next static view. This button simultaneously reflected one of the main starting points that should trigger an emotional response by the participants: We seeded the application with several usability problems, for which we follow the examples by Nielsen [216]. We describe them with respect to Figure 10.1 in the following and highlight the usability problem in *italic* font type.

After the participant starts the study by tapping on ⓪, the U1 screen is presented without any usability problems. The subway line number is always presented in the top left corner Ⓑ; it serves as an orientation support for the observers, when they write down their observations. A tap on ① takes the participant to the U2 screen which features *bad performance* with a loading indicator ⑵ for 5 seconds before the actual text is displayed. Then, ⑵ triggers the next screen. The U3 screen is characterized by *inconsistency* (⑶, ⑶), in a way that the continue button is moved to the bottom and displaying a Chinese text. In addition, a new button with different functionality is added to the same spot used as on other views for the continue button. Button ⑶ allows the participant to navigate to the U4 screen which features *bad performance* and *non-functionality*: they require two taps for on the continue button ④. Hereafter, the U5 screen features *bad performance* and *missing feedback* by imposing a 5 seconds delay between the tap on the continue button ⑤ and the transition to the next view. Eventually, the U6 screen features the same problem as the U4 screen; this time, however, it requires the *continue* button ⑥ to be tapped four times to work, following which the participant will be taken to the final screen. The last screen allows the participant to enter a pseudonym to label the recorded data.

We chose the length of the text describing the respective subway stations to achieve that participants spend approximately 30 seconds to 1 minute per screen. This time frame allowed us to make notes while the participants were busy reading the text—we provide more details on this aspect in the description of the study setting. In addition, with the extended time frame of the participants reading the static content, we attempted to obtain a neutral impression of the participants which does not trigger any emotions. Besides the color of the screens, we design all screens in a similar appearance, to prevent emotions from being invoked on the basis of the screen transitions.

Descriptive Data We performed our study with 12 participants. All of them were either computer science students or academic staff of the Technical University of Munich. Each participant performed the same set of tasks, which was to read the content of the static view and then navigate to the next screen. All of them faced the same usability problems that we seeded within the sample application. The usability problems were programmatically triggered by one or more taps by the participant. The selection of the participants did not follow any rules or criteria toward the individual subjects. The participation was on a voluntary basis. Based on our impression from the contact with the participants throughout each study session, we can rule out the existence of major confounding variables, such as the users' pre-experimental emotion or a stress situation. No further information about the participants was collected. Therefore, to make more assumptions about the impact of person-specific characteristics such as their age, further studies are required.

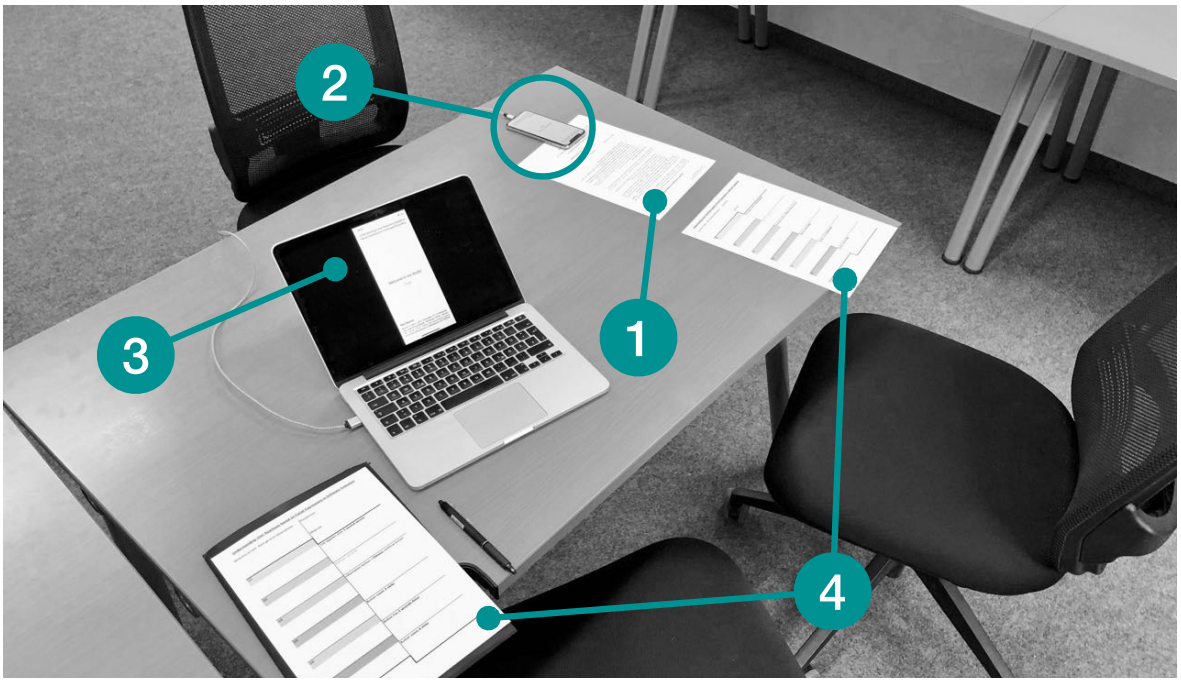


Figure 10.2: The study setting.

Study Setting The study was performed in a seminar room of the Technical University of Munich, the room was blocked to prevent any kind of interruption. We prepared a protocol to describe the study procedure and to ensure comparability between every session with a participant. Figure 10.2 shows a picture of the study setting.

The protocol consisted of three phases.[32] First, we defined an introduction phase in which we welcomed the participants and explained their task, i. e., how they should use the sample application. We further introduced them to the data that we were about to collect, that we observe them while they are interacting with the application, and the overall goal of the study. We closed each introduction phase by handing out a consent which we asked every participant to sign (Figure 10.2-①). We stated that we inform them in case anything is not working out as expected during their interaction with the application. Second, during the conduction phase, the participants started using the sample application that we equipped with EmotionKit to detected and stored the participants' emotional response. The sample application ran on a dedicated smartphone (Figure 10.2-②).

Third and in parallel, two observers sat across the participants and noted down any observation that they considered relevant, such as distinctive facial expressions. The smartphone screen was mirrored to a laptop to get a better understanding of the

[32]We provided supplemental material that is described in the text and depicted in Figure 10.2 in the appendix, i. e., consent, welcome note, and observation sheet in the Appendix D.1, Appendix D.2, and Appendix D.3, respectively.

participants current actions ③. They used an observation sheet ④, which was made up of two columns: the first one provided rows for each individual static subway line view, supported by the color of the view in the sample application to enable easy recognition. The second column provided rows for each individual usability problem, each with a short description. Some example notes included (a) *no reaction*, (b) *smiling*, (c) *hand in front of nose*, (d) *wondering face*, or (e) *twitching cheek muscle*. The observation sheets were merged into one sheet after the study was performed. Mutual consensus after a short discussion was applied during merging of the notes each time the two observers recorded differences. The merged observation sheet later served as the ground truth for the analysis phase in Section 10.1.2 and Section 10.1.3.

Threats to Validity In the following, we present threats to validity of the user study in which we evaluated EmotionKit and its applicability for the detection of emotional responses. We derived a non-exhaustive list of threats from the four aspects of validity, i. e., construct, internal, and external validity as well as the reliability, as described by Runeson *et al.* [265]

Study Setting Even though EmotionKit aims at detecting user emotions in the target environment, the study was conducted in a laboratory environment. Participants were aware of the ongoing experiment and that they were being observed, i. e., that two observers were taking notes about their performance. While one participant explicitly expressed their feeling of being observed, we generally tried to mitigate this threat by creating an atmosphere in which the participants did not feel observed. We can report that most participants quickly reached a focused state in which they were fully focused on the sample application, leaving it very likely that they no longer payed attention to their surroundings. However, for a future advancement of the study and following the suggestion of some participants, we advise the installation of unobtrusive video cameras, which would also simplify the assessment of the observations in form of material that can be repeatably and more precisely be assessed afterwards.

Sample Size The size of the sample of participants represents a weakness of the study that potentially has an effect on the overall results. In particular, a larger number of participants could have improved the accuracy of results and the proof of reliability of the framework. However, this would have complicated the qualitative assessment of the emotional data. We strived for an exploratory approach in assessing and understanding the data to collect a first impression of the possibilities when working with emotions derived from facial expressions that are collected by consumer hardware. Notably, literature claims that a majority of usability problems

can be derived from a comparatively small number of users [219]. This strengthens the analysis of data from a limited population. Furthermore, with respect to the sample of participants, it needs to be stated that all participants had a computer science background. As all of them are directly involved in software development in one or the other way, their feelings toward *bad* usability might not be representative. In general, we can assume that they have a sense for usability problems in mobile applications, such as waiting times or an application crash. We tried to address this threat by designing the sample application in a way that the usability problems are not obvious from the beginning on. On the other hand, the participants knowledge about software development might have affected the overall observation results.

Consent Participants were informed about research field, experiment, and collected data beforehand, due to the university's privacy regulations: By reading the consent, some participants might have discovered the purpose of the study before or during the experiment. We tried to mediate this effect by keeping the study description as general as possible; however, this was not always possible.

Manual Observations We did not ask the participants for self-reported individual emotions, i. e., we did not measure pre- and post-experimental mood of the participants, nor did we record their emotions right after the individual screens which could have been used for precisely determining the performance of EmotionKit, but would also have affected their knowledge toward the study design Therefore, for all aspects, we relied solely on manual observations. As reported as part of the study setting, we used a prepared observation sheet to note down observations during the participants were interacting with the mobile sample application. While this approach potentially reduces the individual precision in the recorded results, it allowed us to ensure and maintain comparability between participants, as we applied the same, subjective measurements to each individual participant. Notably, even if we would have relied on individual reports of participants' perceived emotions, we might have received incorrect, distorted, or biased results [49, 318].

Sample Application Design The study application was designed to induce a set of usability problems, however, it is not guaranteed that they trigger emotions in the participants. Interactions were limited to the simple navigation between static content views, while this limitation could bias the real-world applicability of the results. We tried to mitigate these aspects by designing a sample application that comes close to the typical, text-based applications by relying on standard user interface elements.

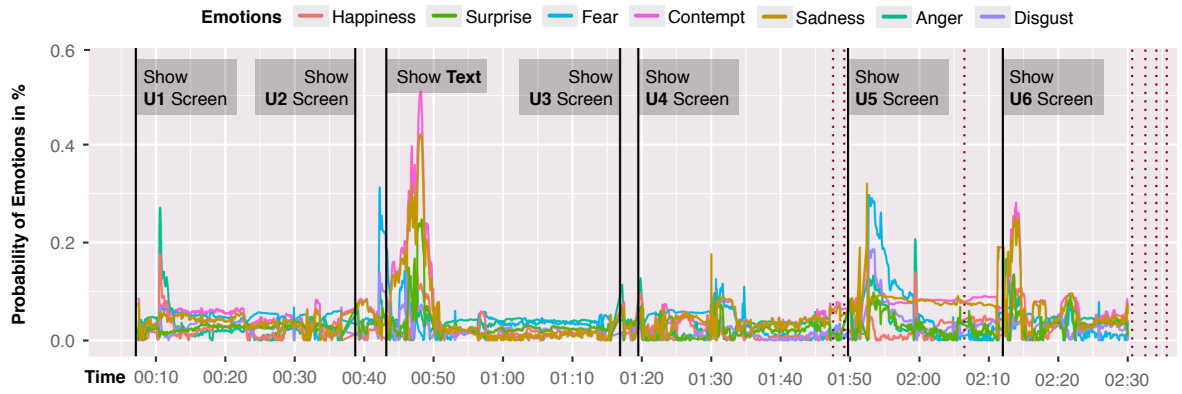


Figure 10.3: Full plot of observed emotions of participant 1 visualized over time. Adapted from Johanssen *et al.* [151] (© 2019 IEEE).

10.1.2 Results of Qualitative Analysis

Through a qualitative study analysis, we aimed to verify the applicability of detecting and recording user emotions from facial expressions using consumer hardware and thereby address RQ 13 and RQ 14.

Data Processing

The sample application creates a log file that contains all recorded data. Each entry includes a timestamp, the current view within the application, probability values for 52 face features provided by ARKit, probability values for all 22 AUs calculated from face features, and probabilities for seven emotions calculated by EmotionKit—as described in the high-level implementation details of EmotionKit in Section 8.2.1.2. In addition, we recorded actions performed by the user, e. g., a tap on a button, with their timestamp. Figure 10.3 depicts a full plot of the recorded time series for one individual participant.

The appearance of a view is marked using a vertical, black line with the view name, from *U1* to *U6*. The delayed appearance of text in *U2* view is highlighted the same way. The user's taps on buttons are represented by red dotted lines. The x-axis shows the time of the recorded emotion; The plot's y-axis shows the change in emotion probability as described below in Equation 10.1.

$$y = \left|\mathrm{emotion} - \overline{\mathrm{emotion}}\right| \tag{10.1}$$

Emotion Analysis

Figure 10.3 depicts the change of emotions of participant 1 in relation to time and the actions performed in the sample application. We compared the observers' notes with the recordings to analyze the emotion. The following description outlines our

analysis results: After the transition to view *U2*, the participant was confused by the 5 seconds delay and appearance of the loading indicator. The participant verbally expressed their confusion by saying *"Ahh .. ok"*. The observers noted the occurrence of *fear* toward the end of the indicator as well as a strong expression of *contempt* and *sadness* after the text had appeared. In a brief conversation after the experiment, the participant stated that they were confused given that the application took a rather long time to load only a small amount of text.

On view *U3*, the participant tapped the button with the Chinese label right away; therefore, they skipped reading the static part of the view. EmotionKit recorded a combination of *anger*, *happiness*, and *sadness* on the transition to both views, *U3* and *U4*. In the observation notes, the observers noted that the participant appeared to be surprised. A change in *sadness* at *01:30* could not be related to the observations. The transition to view *U5* was followed by a change in *sadness* and *fear*. Our observation notes revealed outwards pull from the angles of the participant's mouth. A hand gesture that indicated a questioning state was noted by the observers. The last transition was not recorded, as the participant's hand covered the camera; however, the notes revealed a smile on the participant's face.

> **Summary RQ 13:** Using EmotionKit, we are able to derive users' emotions that matched our manual observations. Some stood out and could be related to usability problems. In general, there is a noise in the recorded data that needs to be addressed when analyzing emotion probabilities for retrieving better results. At the same time, issues in the observation process, such as the participants having their hands next to their chin, impede data collection.

Figure 10.3 exemplifies that the emotions were recorded with a notable time delay before and after the usability problem became visible to the participant, e. g., as it can be seen from the *Show Text* or *Show U5 Screen* screen. We address the aspects of exact time frames with Section 10.1.3.3.

The delays become more visible when presenting and discussing recordings from other participants, as we provide with Figure 10.4. We analyzed three view transitions with their usability problems and compare EmotionKit recordings for the respective three participants with the observer notes. The transitions shown in Figure 10.4 depict individual plots by selected participants which were chosen based on either remarkably similar or contrary patterns. In the figure, each row depicts emotion data plots of a particular transition: In the first row, we show the transition to view *U2* by participants 1, 4, and 12. The second row depicts transitions to view *U4* by participants 4, 5, and 11. The third row details transition to view *U5* by participants 1, 3, and 9. In each of the rows, the emotional data is scaled to the identical time frames along the probability of the given emotion. The color coding is identical to the one which is used in Figure 10.3.

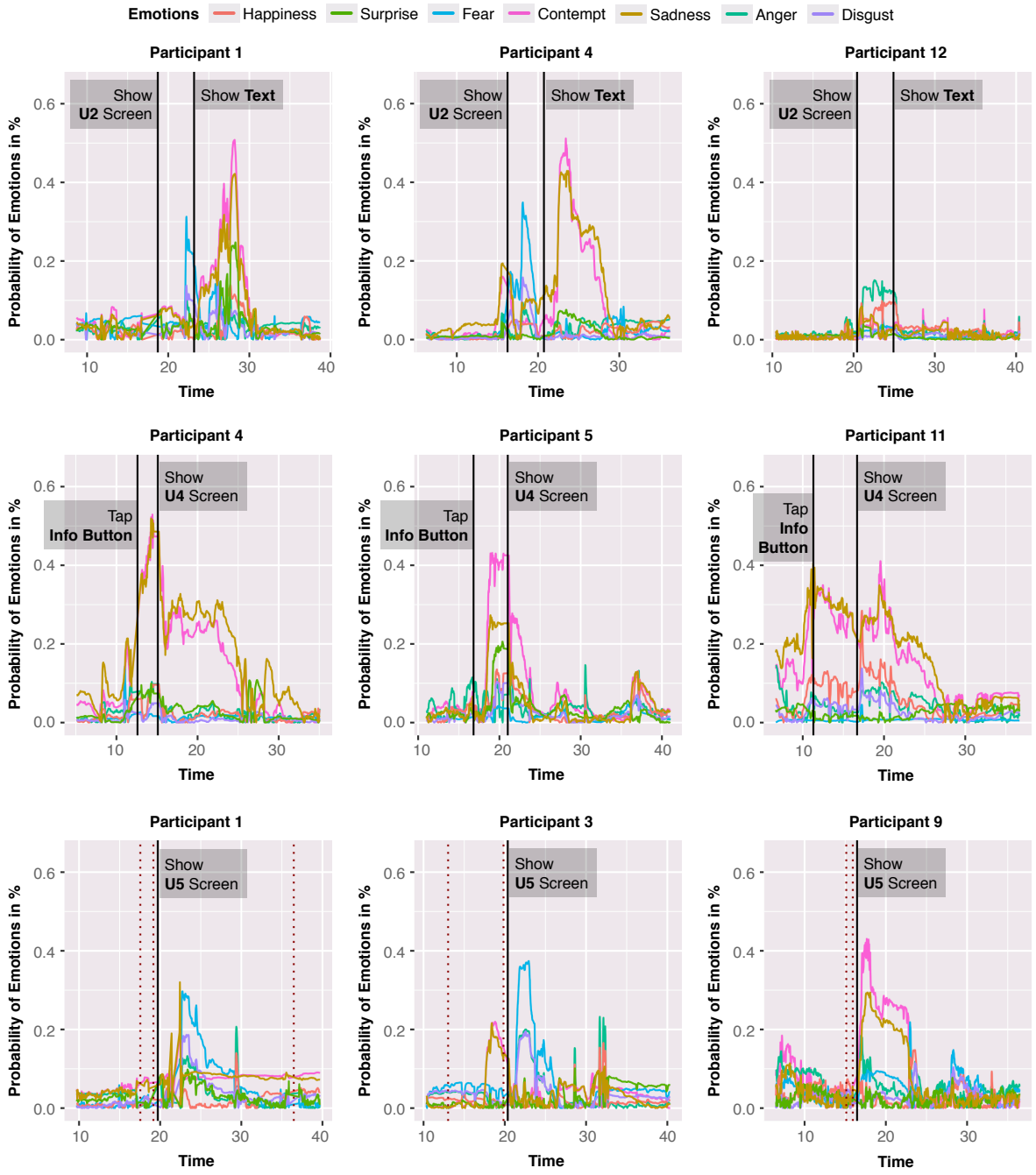


Figure 10.4: Extracts of plots of observed emotions from selected participants.

For the transition to *U2*, participants 1 and 4 both reacted with *fear* while the spinner is showing. After the text appeared, both show a combination of *contempt* and *sadness*. Their emotional response patterns for this transition are remarkably similar: the occurrence of *fear* might be interpreted as a feeling of insecurity, while they were wondering if they did anything wrong. Followed by a rise in *contempt* and *sadness* emotions, they entered a state of relaxation after the text had been shown. This could be understood as a reaction that they recognized it was not their mistake—it was the

system that required time to process. In clear contrast, participant 12 showed some appearance of *anger* for the exact duration of the spinner, suggesting their annoyance about the waiting time: For the remaining nine participants, similar patterns during the waiting time were observed, which, however, were not as distinct.

In the case of the inconsistent user interface and an additional button on view *U3*, right before the transition to *U4*, we found the emotion plots of participants 4, 5, and 11 of high interest. All three participants tapped the *information* button before figuring out what the button with the Chinese text label is used for. Notably, after they tapped this button that triggers a popup, all participants reacted with a combination of *contempt* and *sadness*—similar to the situation that we already observed for participant 1 and 4. After they dismissed the popup and found the correct path to navigate to the next screen, both emotions decreased to their previous levels.

Eventually, we want to highlight the transition to view *U5*, which required the participants to tap on the continue button twice in order to proceed; the last row of Figure 10.4 depicts the respective plots. Participants 1 and 9 both tapped twice quickly as no response happened immediately. After the transition, participant 1 reacted with *sadness*, followed by *fear*. Participant 9 reacted with a combination of *sadness* and *contempt*. Participant 9 showed a stronger reaction than participant 1. We observed a grin with participant 9 after the view transition. Participant 3 waited for about five seconds for a reaction from the app. Missing a response, they reacted with *sadness* and *contempt* before tapping a second time. We observed a grin on the participant's face. In general, as soon as multiple interactions with users are required to trigger an emotion, we found that a comparison becomes more difficult. However, as seen in previous plots, the emotion of *fear* appears to indicate situations in which the user was not sure about what has just happened.

> **Summary RQ 14:** Individual emotion patterns can indicate the occurrence of usability problems in a mobile application. However, not all reactions result in identical patterns. In combination with user interface events, such as view changes or button taps, the observed emotions were identified and utilized to exploit usability problems.

The comparison between emotions raises the question whether we can find dominant traits that reflect particular usability problems. Our study does not provide enough data to answer this question sufficiently. However, we suppose that there are emotions individual to participants which are more prevalent than others: for instance, with respect to participant 3, we observed *fear* as the amplitude that provides the most indications for a reaction, while for example responses of participant 6 are mainly dominated by *sadness*. This supports a hypothesis that emotions cannot be compared between participants, which puts up the demand for an emotion-independent analysis, which we explore in the following section.

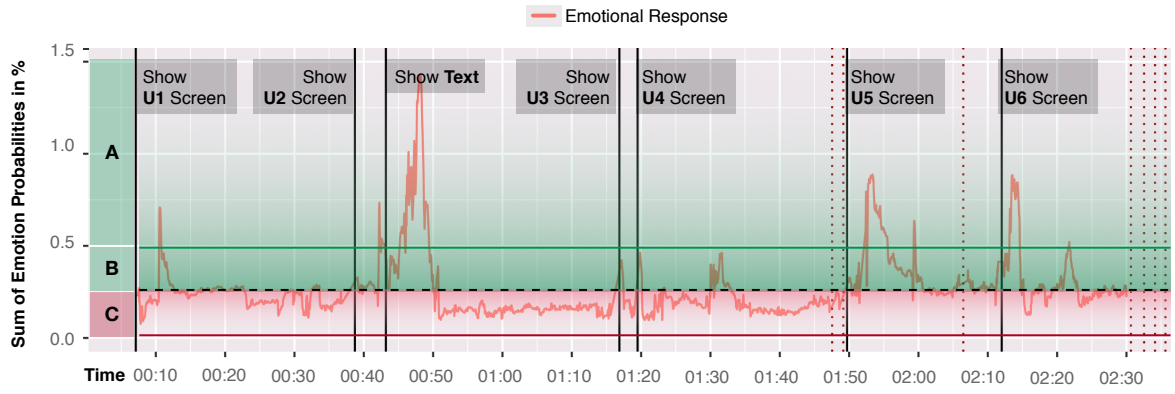


Figure 10.5: Full plot of the emotional response of participant 1 visualized over time.

10.1.3 Results of Quantitative Analysis

From the qualitative analysis, we learned that it might not be the moment of occurrence, but the time span afterwards that indicates a usability problem. This, in turn, might pose the risk that we confuse the usability event with the view change event. On the other hand, the fact that it started even before the other window was shown supports the relationship of a response toward the usability problem at hand.

We performed a second, more detailed analysis of the recorded emotions to investigate all participants' facial expressions toward the usability problems in a quantitative approach. In order to clearly detect the emotional reactions, we combined the individual emotions in one amplitude to create a signature that reduced the noise sensitivity for the recordings. We describe the process of data preparation in Section 10.1.3.1 and present results of a binary classification in Section 10.1.3.2. In addition, we provide a fine-grained analysis of time frames in Section 10.1.3.3.

Data Processing

As described by Equation 10.2, we sum up all seven emotions, i. e., happiness, sadness, surprise, anger, fear, disgust, and contempt, and derived a new value which—in contrast to Equation 10.1—can exceed 1.0. The summation allows for a simplified identification of changes. We refer to this value as the **emotional response**:

$$y = \sum_{k=1}^{7} (\left|\mathrm{emotion} - \overline{\mathrm{emotion}}\right|)_k \tag{10.2}$$

In Figure 10.5, we plot the emotional response of participant 1, which we already showed in Figure 10.3. The emotional response allows to clearly observe peaks in the recorded data, while the overall response during reading phases, i. e., phases with a neutral face, settles down to a steady and low level, which reduces the noise that is present in Figure 10.3.

For the assessment, we horizontally split the plot into three ranges: Peaks in range **A** clearly refer to an emotional response. Reactions in range **B** are likely to be treated as an emotional response, however, their context—which is a reasonable time frame before and afterwards—should be considered. We interpret range **C** as recordings that indicate no emotional response.

Binary Classification

Based on processed data, we can describe the output of EmotionKit as the result of a binary classification to evaluate its performance:

TP is a true positive classification, in which the framework detects an emotional response that has also been recorded by the observers.

TN is a true negative classification, in which neither the framework nor the observers were able to detect an emotional response.

FP is a false positive classification, in which the framework detects an emotional response; however, the observers do not record an emotional response.

FN is a false negative classification in which the framework does not detect an emotional response; however, the observers do record an emotional response.

We relied on the observer notes to identify the classes for the actual emotional responses. Subsequently, we manually created emotional response plots for every participant—as we exemplified in Figure 10.5 for participant 1—and manually inspected the emotional response to derive the predicted class. We summarize the outcomes of the two-class prediction in Table 10.1. The table follows the structure of the sample application introduced in Figure 10.1: Column titles ending with a "C" represent a static content view; e. g., **U1-C** describes results for screen of U1. Column titles ending with an "I" represent the application's interactive part containing the usability problem; e. g., **U1-I** describes the results for *bad performance* as denoted with (2a) in Figure 10.1.

Following this classification, we can describe the performance of EmotionKit using both the *Sensitivity* and *Specificity* values, as well as the *Accuracy* [312]: The sensitivity describes how many actual emotional responses by participants have been correctly detected as an emotional response; this is known as the *recall* or the *true positive rate*. The specificity describes the number of occurrences in which a participant did not show any kind of emotional response and that was actually detected as a non-emotional response by EmotionKit; this is known as the *true negative rate*. The accuracy summarizes how many instances in total have been detected correctly. In Table 10.2 we list the results, while we provide a separated overview of a combined value as well as content and interaction values only.

Table 10.1: Binary classifier outcomes of the EmotionKit performance, adapted from Johanssen *et al.* [151] (© 2019 IEEE).

#	U1-C	U1-I	U2-C	U2-I	U3-C	U3-I	U4-C	U4-I	U5-C	U5-I	U6-C	U6-I
1	TP	TN	TP	FP	TP	TP	FP	TN	ERR	TP	FP	ERR
2	TP	TN	TN	TN	TP	TP	TN	TP	FP	TP	FP	TP
3	FP	TN	TN	TN	ERR	FP	TP	TP	FP	TP	TN	FP
4	TN	TP	TN	FP	TN	TP	TN	FP	TN	TP	TN	ERR
5	TN	FP	FP	FP	TN	TP	TP	TP	FP	FP	TN	TP
6	TN	FP	TN	FP	TN	TP	TN	FP	TN	FP	TN	TN
7	TN	TP	FP	TN	TN	TN	TP	TP	TN	TP	TN	TN
8	TN	FP	TN	FP	TP	FP	FP	TP	FP	TP	FP	TP
9	TP	TP	TN	TP	TP	TP	TN	TP	TN	TP	TP	TP
10	TP	TN	TN	ERR	TP	FN	TP	TN	FP	ERR	FP	ERR
11	ERR	FP	TP	TN	ERR	TP	TN	TN	TN	TP	TP	TP
12	TN	FP	TN	FP	TN	FP	TN	FP	TN	TP	TN	TP

Given the results in Table 10.2, it can be stated that EmotionKit is able to detect the emotional responses of the participants (0.98 for the combined analysis; 1.0 for content; and 0.97 for interaction). The correct detection of non-emotional responses is more challenging as indicated by the results for specificity: Non-emotional responses are detected moderately for content (0.73), however, detection during interaction is low (0.39). The accuracy states that three out of four instances of emotional responses are detected correctly in a combined scenario using EmotionKit.

As depicted in Table 10.1, we observed 16 emotional responses and 38 non-emotional responses during the presentation of the static content views, while the participants responded with 34 emotional and 13 non-emotional responses to the interactive parts of the sample application. These data suggest that the study design follows its intention, i. e., participants are less influenced by the content and we are likely to observe the emotional responses related to the usability problems. The

Table 10.2: Sensitivity, specificity, and accuracy values of the study, adapted from Johanssen *et al.* [151] (© 2019 IEEE).

	Combined	**Content**	**Interaction**
Sensitivity	0.980	1.0	0.9706
Specificity	0.600	0.7308	0.3939
Accuracy	0.7407	0.7941	0.6866

observation that 10 out of 12 participants in *U2-I* did not show any emotional response to the interaction itself eliminates the assumption that emotional responses are triggered by the transition—note that *U2-I* was the only interaction without any seeded usability problems.

More than one-fifth (23.61%, resp. 34 occurrences) of the observations were classified as *FP*. Two observers individually collected and mutually agreed to the observation notes, however, they are not trained experts in reading emotions. As the expressions are known to be sensitive to changes, probably many of the classified *FP*s may actually be *TP*s. Therefore, the real rate remains unclear. However, since we detect more than two-third of correctly classified occurrences (69.44%, resp. 100 occurrences), we consider the false classification of *FP*s as a subject for further research. In fact, a high ratio would support the need for technology support and further highlight the benefits of EmotionKit.

Summary RQ 15: Participants show a higher emotional response toward the interactive parts than to the static content. Our observations suggest that this behavior is related to the seeded usability problems. At the same time, we noted that the occurrence of emotional responses can be detected better than the absence of emotional responses.

We classified some occurrences in Table 10.1 as an error (*ERR*), which we did not incorporate in the analysis above. We defined an error as a situation in which EmotionKit could not record any data or the data was noisy because of a reason that was observed by the observers.

In Figure 10.6, we present three examples of classified errors that we experienced during the study. Ⓐ shows the plot of participant 1 (*U6-I*): the data at the end was cut off since the recording was stopped too early. Ⓑ shows the plot of participant 3 (*U3-C*): the peak relates to a situation in which they had their hand in front of their face and moved the head back and forth. Ⓒ shows the plot of participant 11 (*U1-C*): the participant flipped the device around and put it down on the table to ask a question; the emotional response, plotted on the right side, clearly points out the error Ⓓ.

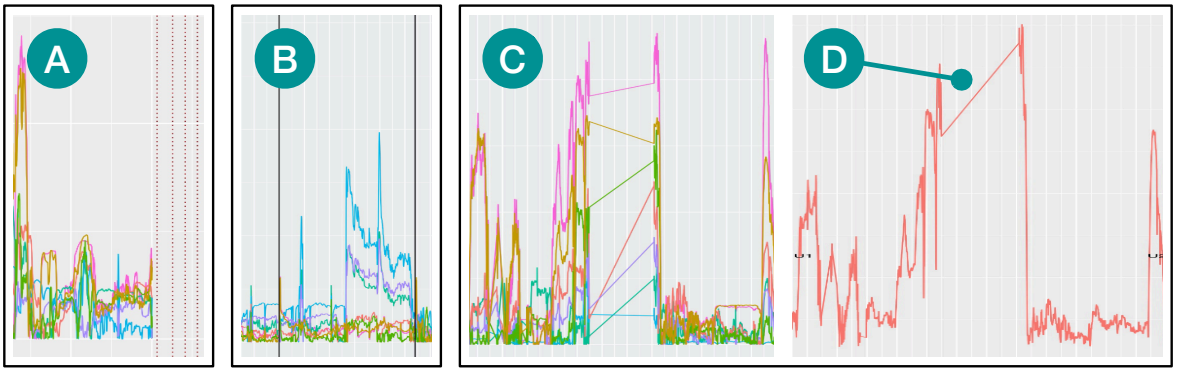


Figure 10.6: Three examples for errors during the study.

We understand the total of nine errors (6.25%) as a good value for the performance of EmotionKit supporting its applicability. However, as the data resulted from a laboratory study, we assume that the error rate will be considerable higher in real-world scenarios, given varieties to hold devices or other *external stimuli* [101].

Phases of Emotional Responses to Usability Problems

To better understand the emotional response toward usability problems, we analyzed the time series between two static content views and found three phases of interest, as depicted in Figure 10.7.

For the calculation of the time frames, we processed the data as follows. First, we only considered data from interactions that contained a usability problem, which removes time frames of *U2-I-U3*. Second, we only considered data from interactions that were recorded as a *TP*. Third, since we did not specify a clear end point of *U6-I*, we also truncate the corresponding observations.

Overall, we considered 27 interactions, for which we measured three time frames as depicted in Figure 10.7. Ⓐ describes the time of delay until we recorded an emotional response to the usability problem. Ⓑ describes the time until the emotional response reached its peak value. Ⓒ refers to the time recorded until the participant reached a *neutral* state which can be clearly related to consuming the next static view. This information may be used to automate the detection of emotional responses; the time frames could be used to create heuristics for differentiating between noisy data and actual emotional responses.

We found that participants on average showed an emotional response after 2.183 seconds after the usability problem occurred. After an additional 2.987 seconds, on average, they reached a peak on emotional response that indicates the highest measured point of the amplitude. Hereafter, it took participants on average 7.073 seconds to return to a neutral state in which no emotional response could be measured.

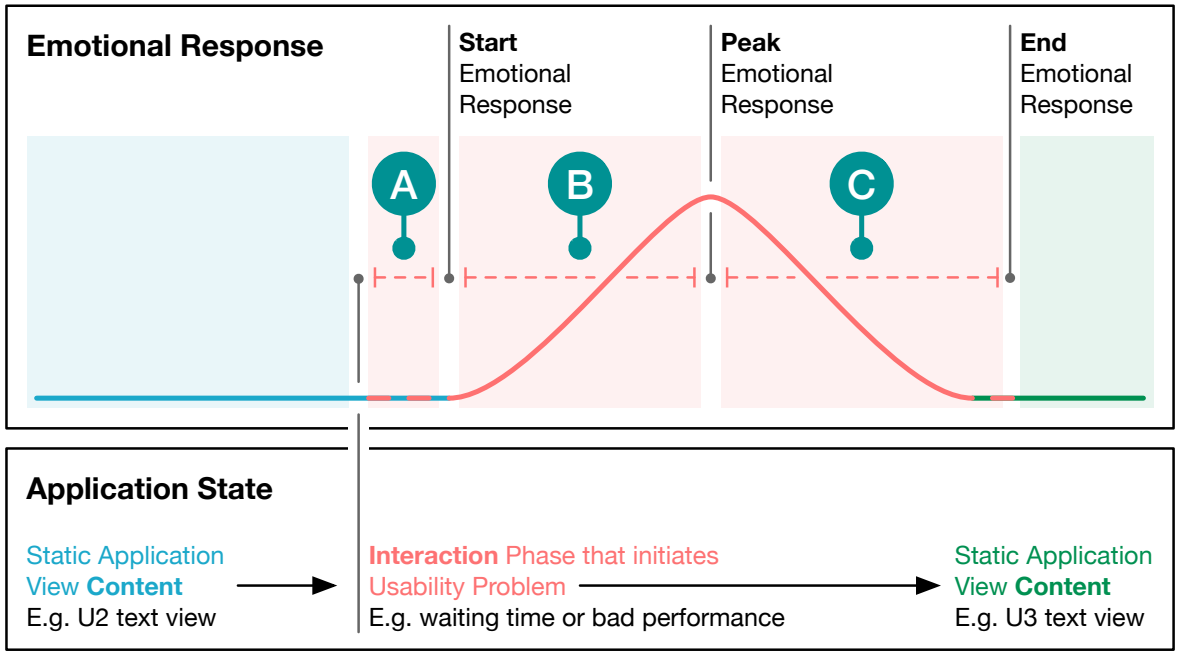


Figure 10.7: Phases of emotional responses.

Summary RQ 16: The emotional response can be split into three phases: the time until the start, peak, and end of an emotional response. On average, we measured these phases with approximately 2, 3, and 7 seconds, which sums up to a time span of 12 seconds for an emotional response toward a usability problem.

10.1.4 Discussion of Results

We present our interpretation of the validation by discussing on the implications of the results for future evaluation and application in practice. Our observations suggest that tracking down emotions from users' facial expressions can support the detection of usability problems during interaction with a mobile application.

Understanding Emotions

Users react in different variations to the same usability problems. In our study, not all participants showed visible reactions to some or all problems, and in real-world scenarios, other factors—such as the cultural background toward visible reactions include confusion and surprise, as well as humor like laughing and smiling—might influence users' facial expressions. We did not find participants reacting to the same usability problems in one way only, but the plots indicate similar emotional patterns per participant. Hence, developers are confronted with uncertainty when trying to understand the emotional reaction of multiple users to the same interactive element, leaving them with a need for help when making a decision on such a basis.

In contrast to the spot widgets presented in Figure 8.6, the validation results point toward the usage of individual emotional response plots as depicted in Figure 10.5. This would allow to facilitate the discovery of sudden changes for a short period of time, since such changes typically indicate unexpected software behavior. This is supported by the assumption that longer lasting emotional responses are more likely to relate to the application content. This is further emphasized by the fact that reading emotions requires psychology domain knowledge that a developer may lack; for example, a grin can be recorded as a combination of *contempt* and *sadness* as it becomes obvious from the plot in Figure 10.3. We were able to understand the EmotionKit's recordings given the help of our manual observation sheets.

To improve emotion understanding and make them accessible to developers—with the permission of the user—we envision to record a short video when an emotion peak is detected. This might help developers to assess the observed emotions in retrospective, similar to the observation notes used in the study. This functionality may be encapsulated by a new knowledge source.

As a long-term prospect and given an improved maturity level of the analysis of recordings, EmotionKit might also support experts from other domains in understanding their users in respective tasks. However, this requires studies to confirm the sensor data reliability, explore parameters in the calibration process, and comparisons with other input sources, such as a front-facing mobile camera.

Creating Relations to Software Increment

As it becomes obvious from the validation results, additional information about the application under observation is indispensable for interpreting the data. In the case of our sample application, the recorded time stamps for the *view change* events, *show text* event, or the *user tap* events contributed considerably to the understanding of the observed emotion and the alignment of the manual observation notes. Therefore, integrating EmotionKit with other knowledge sources can support developers to better understand the reason for a user emotion: Using feature crumbs that are recorded by FeatureKit support developers to put the spotlight on parts of their application where they expect usability problems to occur. This helps to analyze the emotional response. Furthermore, additional research should evaluate whether a particular requirement that addresses usability aspects can be mapped to a specific emotional response.

Automating Peak Detection

We suppose that metrics about the observed emotions can be created to support the automatic identification of situations in which usability problems might have occurred. For example, similar to the proposal described by Begel [30], a peak of *fear*,

followed by either a peak of *contempt* or *sadness* may be treated as a sign for a *bad performance* issue. Similarly, tracking a full spectrum of emotions as suggested by Rozin and Cohen [264] may help to notice only the relevant changes in user feelings toward the software. We see the processing step presented in Section 10.1.3.1 as a similar step to retrieve a unique spectrum of emotional response that can be utilized for further processing. Likewise, the time frames derived in Section 10.1.3.3 can be used and—in combination with the definition of a threshold—serve as an additional trigger for a usability problem indicator. Developers can be notified by a system, such as **Cuu** workbench, as soon as such usability problem indicators are detected. For instance, they could receive automatic notifications as soon as emotional changes to their latest increments are detected.

10.2 Toward Validation of other Kits

The validation of EmotionKit can serve as a blueprint for validating the remaining kits. In this section, we provide suggestions on kit-individual aspects for a validation as well as—if available—initial and explorative results.

Individual Validation

We detail individual validation approaches for the remaining three kits, namely ThinkingAloudKit, BehaviorKit, and PersonaKit.

ThinkingAloudKit

Regarding ThinkingAloudKit, we suggest to conduct different types of experiments.

Performance First, as the overall results of ThinkingAloudKit depends on the accuracy of the underlying speech-to-text and classifier performance, we suggest to further analyze different frameworks that allow for the transcription of speech as well as compare the results of multiple machine learning approaches. From our experience, speech-to-text frameworks are not optimized for transcribing loosely coupled text fragments. This, however, is typical for collecting Thinking Aloud protocols. On the other hand, with respect to the classification of results, we created a classification prototype for Continuous Thinking Aloud (CTA). Therefore, we relied on 10 manually collected and transcribed Thinking Aloud protocols, from which we used 200 sentences as the training data set. Using *Bag of Word* dictionaries combined with a logistic regression classifier for classifying the sentences, we achieved a testing accuracy of 88%. The small data set can be understood as a limitation of the implementation of ThinkingAloudKit. We argue that it fulfills the purpose of a concept proof, however, a larger data set and more advanced classifier approaches

are required to enhance the results. In addition, multiple attributes need to be considered when training the machine learning classifier. For example, the choice of protocols needs to be further considered, as they directly influence the classifier results. For instance, individual traits that become visible from the verbal descriptions might bias a classifier. Another example might be differences in languages; not only do the language differences prevent usage between languages, they might also affect the way users describe their thoughts.

Feasibility As part of an additional laboratory experiment, we suggest to test the feasibility of ThinkingAloudKit by comparing its results with traditional approaches of collecting and understanding Thinking Aloud protocols. This also includes the collection of feedback from developers on how they perceive the usefulness of the widget. We already collected first insights as follows. First, we compared five sessions that were captured using ThinkingAloud kit and five traditional Thinking Aloud sessions with a total of ten test subjects. Initial results revealed that performing a CTA session is on average more than 3.5 times faster; this does not yet include the time required for transcribing the recordings in case of traditional Thinking Aloud. One potential reason for the increased speed might be the lack of a supervisor that could be consulted for questions during the session. Second, we showed the widget visualization to six developers, all of which were able to identify crumbs with potential usability problems. However, two feature crumbs that did not contain any problem at all were also considered relevant. This insight starts the discussion if more roles besides developers are required in the process, i. e., a dedicated role for both setup and analysis of CTA results as depicted in Figure 8.7.

BehaviorKit

Given the results of BehaviorKit summarized in Table 8.1, in the following, we outline major implications as well as limitations with respect to the way we collected data for the classifier development as described in Figure 8.9.

Implications The presented results provide a first indication that it is possible to infer *usage-related* and *application-related* information based on behavioral usage data. We conclude that our approach for inferring the *user-related* information does not work, which is in particular founded in the available dataset; we argue that this aspect needs to be addressed in more detail using a laboratory experiment.

Limitations We developed multiple classifiers that provide insights about the users based on various input parameters. We trained the classifiers as described in Section 8.4.1.3 and Section 8.4.1.1, which pose limitations regarding their validity.

We restricted data collection to one week, which results in a limited dataset. Small datasets tend to have the problem of overfitting [36]. Consequently, the validation set is rather small and limits the generalizability of the results. Likewise, the selection of participants might not reflect actual distributions of users, which has an influence on the accuracy. The data used to train and evaluate the classifiers were collected using only one particular application. Along with the fact that we only used one platform, i. e., Apple iOS, for the data collection, the generalizability of the results is also limited to this application. However, given that the application was chosen because it makes increased use of typical interaction elements, it needs yet to be investigated whether the results are generalizable or individual to the application at hand. Moreover, consciously designing the data collection in a way to induce certain states such as usability issues may have led to an over-representation of some states and characteristics. Also, the active categorization that was done during the training data collection relied on subjective human judgment by the study observer.

PersonaKit

With respect to the runtime personas presented by PersonaKit, one of the most interesting aspects of validity is the usefulness for developers in working with the widget. Therefore, we suggest to collect more insights on what information should be presented and how. We propose the utilization of a survey as an empirical strategy. Following the overall validation of **Cuu**$^{\textbf{SE}}$, we already received valuable feedback from managers; for instance, they state additional characteristics that would be of use for them, as well requesting different presentation modes, such as showing the runtime persona's attributes in a structured form, rather than a block of text.

Interrelated Validation of Kits

Every **Cuu** kit provides its own expertise in order to process usage data to usage information as well as extract usage knowledge. So far, we only addressed the individual validation of a kit, answering *effect questions*, i. e., answering the knowledge question whether a treatment produces a desired effect in a certain problem context [310]. Wieringa notices that, as part of the treatment validation, there are more questions to be answered [310]. This includes *trade-off questions*, i. e., the investigation whether one artifact for a treatment performs better, worse, or similar than another one, provided the same problem context, as well as *sensitivity questions*, i. e., addressing the produced effect of an artifact assuming that the problem context changes [310]. We discuss starting points toward answering these knowledge questions.

Trade-off Questions

With respect to the trade-off between kits, Table 10.3 provides a non-exhaustive list of attributes which could be used for comparison.

Table 10.3: Non-exhaustive list of attributes for address trade-off questions.

Attribute	EmotionKit	ThinkingAloudKit	BehaviorKit	PersonaKit
Depends on other kits?	FeatureKit	FeatureKit	FeatureKit and InteractionKit	FeatureKit and InteractionKit
Requires sensor permissions?	Yes (3D Camera)	Yes (Microphone)	No	No

The less dependencies a kit exhibits, the better; while all kits depend on the simultaneous activation of FeatureKit, some kits such as BehaviorKit and PersonaKit also require InteractionKit. On the other hand, EmotionKit and ThinkingAloudKit require additional permissions from the user to access the 3D camera and the microphone of the devices, respectively. As the acceptance of in particular biometric sensor may differ from user to user [324], this may decrease usability of both kits in a similar context compared to BehaviorKit or PersonaKit.

Sensitivity Questions

With respect to the sensitivity of kits to the problem context, Table 10.4 provides a non-exhaustive list of attributes which could be used for comparison.

Table 10.4: Non-exhaustive list of attributes for address sensitivity questions.

Attribute	EmotionKit	ThinkingAloudKit	BehaviorKit	PersonaKit
Type of user feedback	Implicit	Explicit	Implicit	Implicit
Compatible with only one user?	Yes	Yes	Yes	No

We designed **Cuu^{SE}** with a focus on usage with a low number of users; however, this number may change and the kits may cope this increase differently. Except for ThinkingAloudKit, all kits consider implicit user feedback, which generally does not limit the number of users; note that ThinkingAloudKit uses automatization to cope with this aspect. At the same time, while EmotionKit, ThinkingAloudKit, and BehaviorKit deals well with only a single user, PersonaKit requires at least two users to classify different runtime personas.

Chapter 11

Design and Validation of a Syllabus

> *"Software engineers lack familiarity with many of the issues of human factors in user interfaces, partly because they and their peers are highly skilled users, and so they make most usability design decisions unconsciously, unaware of the possible implications. This problem is serious because many user interface design decisions are left to the discretion of software engineers, an area of design of little interest [which] should not be taken as a lack of concern; I don't know of any software developers who want a bad user interface."*
>
> — GARY PERLMAN [243]

As we are moving toward the validation of the **Cuu**^{SE} framework, this chapter is concerned with the goal of providing an instrument to disseminate **Cuu**^{SE} to developers. We introduce the **Cuu** syllabus that enables the introduction of the core **Cuu** artifacts within a multi-project course in an academic environment. The syllabus focuses on hands-on utilization of **Cuu** artifacts and combines this with teaching concepts of usability engineering, as they form the basis for user understanding.

The chapter is structured as follows. In Section 11.1, we describe the need for a the syllabus and outline related work. We introduce the iPraktikum in Section 11.2 which serves as the target environment for the syllabus. We describe the concept of cross-functional teams that allow to transfer knowledge into project teams, which reduces the need to do this with every individual team member. In Section 11.3, we detail the structure of the syllabus as a means to provide teaching support for instructors. This encompasses four meetings and a course-wide lecture. *"[T]reatments need instruments, and these need to be validated on their effectiveness."* [310]—with the instrument being the **Cuu** syllabus, we validate its applicability in Section 11.4. Therefore, we assess observations from the utilization of the syllabus over the course of a semester. These insights also supported improvements of the **Cuu** artifacts.

11.1 Instrumental Design

We set out the Instrument Design Goal which shall help to answer the knowledge question on the validity of **Cuu**$^{\mathbf{SE}}$: We derived a concrete instrumental design problem in Section 1.4.2, which we resolve with the design of the **Cuu** syllabus. While we focus on the context of an academic environment to address this problem, the syllabus is also applicable for industrial settings. This is based on the assumption that we can rely on students as a valid simplification for professionals within laboratory experimentation [98].

The core question to answer is how can we teach students to make use of feature crumbs, the **Cuu** workbench, **Cuu** kits, and the **Cuu** workflow as introduced in Part III. As pointed out by practitioners in Chapter 4 and our observations in Section 7.2.2, this is needed as designing the correct feature path is similar to writing effective test cases: it requires experience and knowledge.

11.1.1 Challenges

There is an acknowledged need to put a stronger emphasize on the combination of agile development and user research [61]. This is enforced by the availability of software for a great range of platforms, reaching from desktop computers to mobile devices: the need for teaching usability engineering gained high importance. However, there are two main challenges that impede the adoption of usability engineering in software engineering classes, as we describe in the following.

First, many software engineering classes focus their teaching efforts on development practices as well as engineering theory, while they miss out on addressing the usability aspects of an application—which results in a situation in which developers design user interfaces [244]. Nevertheless, students usually have a general understanding of usability engineering that raises from different sources. On the one hand, this is based on common knowledge, intention, or personal interest in the topic. On the other hand, there are lectures, that either are focused on teaching usability concepts to the full extend or discuss usability engineering as a subtopic [180]. In most of the cases, however, students lack a real-world scenario in which they can apply new usability engineering knowledge to real-world applications.

Second, many usability engineering concepts require a well-defined set of tools, requiring an environment which is not easy to provide. For instance, even though it is considered a discount usability technique, the *Thinking Aloud* protocol by Nielsen [216] requires the preparation of the application in order to perform a thinking aloud session; this does not scale in case it needs to be applied multiple times, and it is further not easy to apply in the given time frame of a semester, even when the students know how to do it. Finally, it requires more steps beforehand, such as creating an

understanding of the feature under development, prepare it for testing, defining how its performance can be measured, and delivering the feature to a test user.

To approach both problems, we argue that students require a well-aligned teaching concept as well as tool support to enhance the effectiveness of teaching usability engineering. In addition, other researchers stress the need for hands-on experience and constraints from industry [61]. Therefore, we are convinced that such a teaching concept needs to be set within a project course that provides a hands-on environment to apply usability engineering concepts over a considerably longer timeframe in a practical manner, as already suggested by Wohlin and Regnell [313].

As a result, and in order to address the instrumental design problem described in Section 1.4.2, we designed the **Cuu** syllabus that enables the dissemination of the **Cuu**$^{\textbf{SE}}$ framework with the overall goal to provide a teaching approach for continuous user understanding. We build upon the assumption that the combination of both theoretical concepts and practical elements helps to make the usability engineering more tangible for students [216]. The following descriptions are also motivated by the idea of encouraging interaction and collaboration when teaching usability engineering as suggested by Carroll and Rosson [57]. Furthermore, we foster transparency by using collaborative tools and interactive teaching units that have a high dependency on applications that are developed by the students themselves.

11.1.2 Related Work

With the **Cuu** syllabus, we follow the advice by Jakob Nielsen on teaching usability engineering by *"bas[ing] the course firmly in the laboratory"* [216] and thus teaching user understanding in applied, project-based courses to prepare students for their later careers [103, 228, 313]. Nielsen stresses that—besides learning a basic set of skills through theoretical lessons—a hands-on approach is indispensable for students to learn and understand the concepts of usability engineering, but also to trigger a "revolutionary change in [their] attitudes" [216]. This hands-on approach is further supported by other researchers, such as Basili *et al.* and Ghezzi *et al.*, who stress that students need to apply their theoretical knowledge in practical projects [25, 119] in order to benefit from them [38, 65, 143, 289].

Offering students the chance to acquire the basic skills motivated us to support the **Cuu** syllabus with theory sessions at the beginning of individual meetings, which are then followed by hands-on exercises. According to Nielsen, this experience is in particular valuable if the students investigate the usability of applications they developed on their own [216]—we follow this theme by introducing tasks for homework that relate to self-developed applications. Chan *et al.* present research on the integration of teaching human-computer interaction aspects into the curriculum of master classes [59]. They highlight the importance of real customers and ongoing discussions [59]. Therefore, we describe a syllabus that integrates usability engineer-

ing in a project course as described in Section 11.2, in which the students develop applications for real customers from industry.

As stated with the opening quote of this chapter, the need for teaching usability engineering was already highlighted by Perlman in 1988 [243]. The fact that he continued to work on teaching usability engineering in 1995 [244]—almost 10 years after his initial work was published—shows that teaching usability engineering needs both continuous refinement and adjustments over time. By adding agile concepts and incorporating state-of-the-art technology into our syllabus, we try to ensure its compatibility with recent technologies.

Newer research of teaching usability engineering is presented by Ovad *et al.* [232]: they try to enable developers in an agile industry setting to perform basic usability tasks, such as A/B tests, on their own [232]. Bruun *et al.* use a similar approach in an industrial setting to teach developers the concepts of usability evaluation. With only three participants, the level of generalizability is limited, however, the results of developers quickly learning the core concepts of usability engineering look promising [50]. As shown by these researches and supported by others [218, 220], experienced developers are often the audience in receiving new insights on usability engineering.

11.2 The iPraktikum

The syllabus targets its utilization within the iPraktikum, a capstone course [46], which provides the environment for teaching usability engineering in an agile and multi-project course structure, which we describe in this section. We further outline the role of cross-functional teams.

11.2.1 Course Structure

The iPraktikum is a multi-project course which is run every semester with 70-90 student developers who work in up to 12 project teams to create an application within a mobile context for real customers from industry in order to solve their real problems [46]. While the mobile context is realized using Apple's iOS platform, resulting in iPhone and iPad applications, most projects are not standalone solutions. They include application servers, sensors, actuators, or wearable devices.

As shown in Figure 11.1, each *Project Team* consists of a *Customer* from the industry, a *Project Management Team* consisting of a *Project Lead* and a *Coach*, as well as the *Development Team*. Overall, the iPraktikum consists of multiple project teams that are composed of different roles and sub-teams. Both the project lead and the team coach fulfill a role similar to a scrum master. The project lead is a teaching assistant who is experienced in leading projects. The coach is a student who already partici-

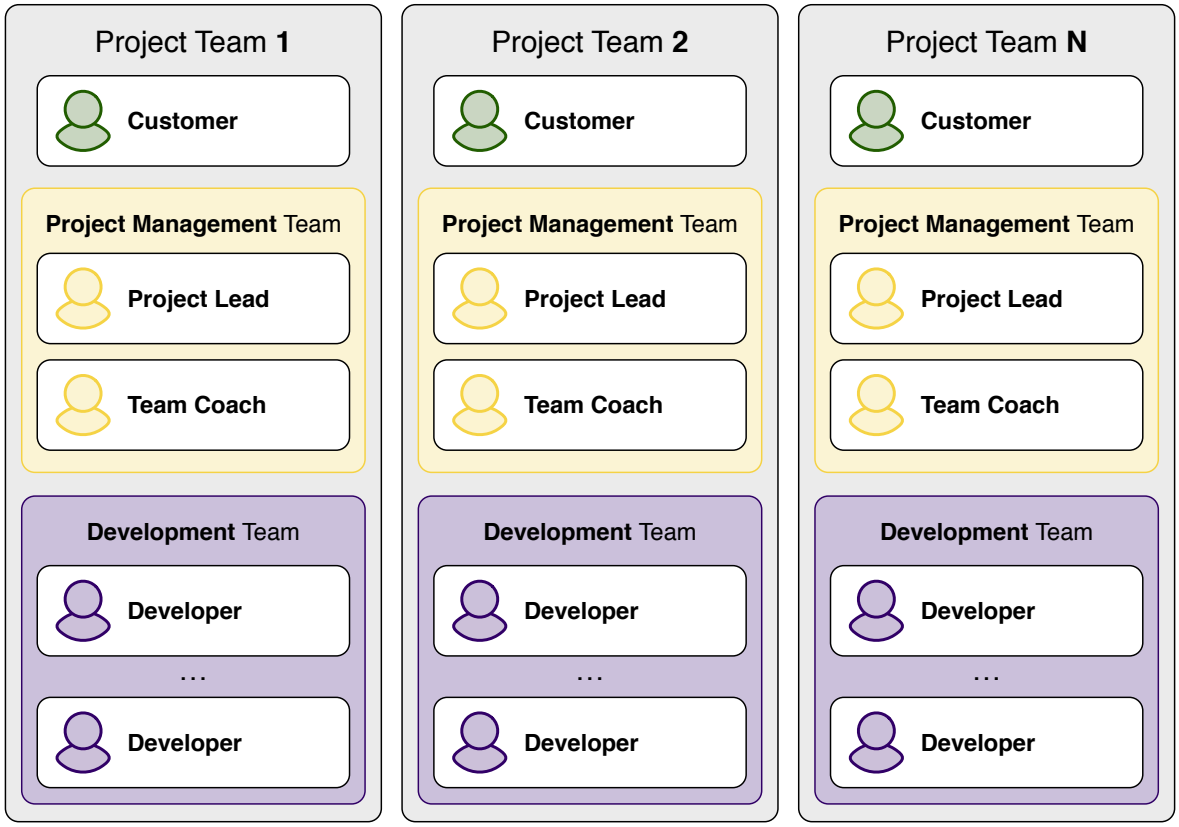


Figure 11.1: The iPraktikum team structure, adapted from Johanssen *et al.* [152].

pated as a developer or can provide work experience in managing teams. Thus, the project management team members are familiar with the infrastructure and teaching methodology. The *Developer*s are students from second Bachelor semester up to their last Master semester.

The iPraktikum fosters an event-based methodology, which enables the students to continuously interact with the customer [176]: it implements the idea of CSE as described in Section 3.1. Furthermore, the iPraktikum emphasizes the development of early prototypes and their vivid and theatric presentation to customers [316]. Over the course of multiple semesters, it has been shown that the format of the iPraktikum as well as its processes contribute to an increase in the skills of students regarding both technical and non-technical aspects [46].

11.2.2 Cross-Functional Teams

The iPraktikum relies on cross-functional teams [46, 176, 316] in order to focus on special topics of relevance. Each of these teams is run by one or multiple cross-functional coaches and led by a cross-functional instructor, as shown in Figure 11.2. Cross-functional coaches are students experienced in the respective field. A cross-functional instructor is a teaching assistant who ensures the teaching methodology.

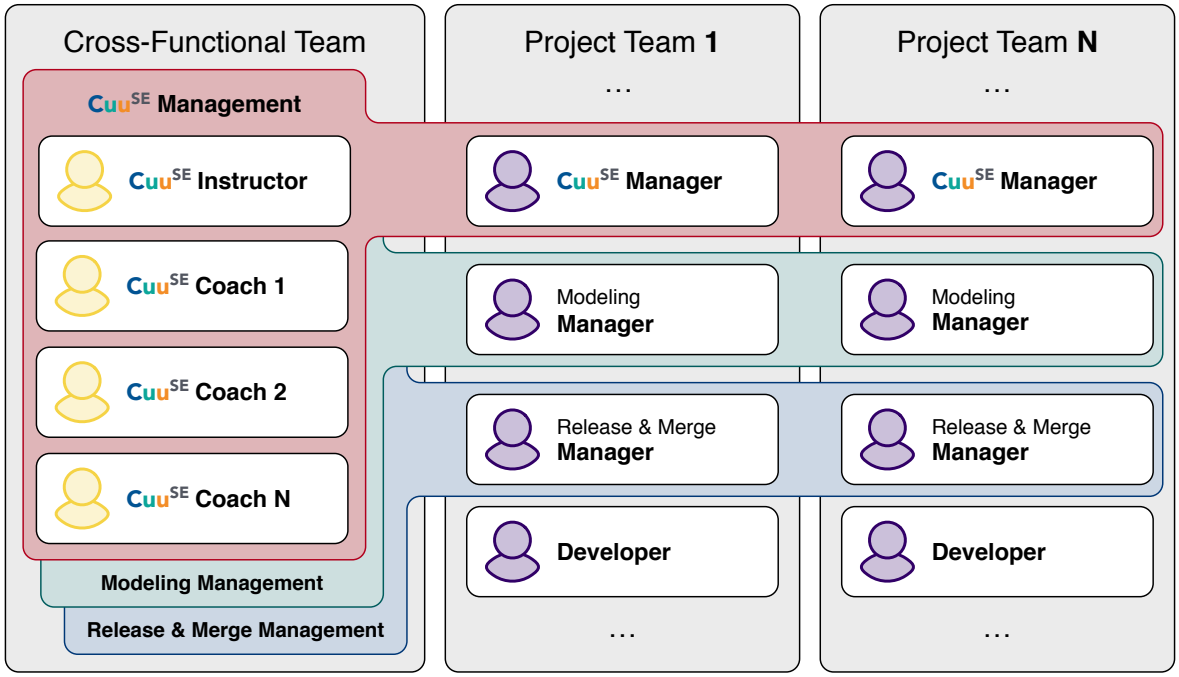


Figure 11.2: The iPraktikum cross-functional team structure, adapted from Johanssen *et al.* [152], focusing on the team for continuous user understanding.

The cross-functional coaches organize work with the cross-functional teams almost independently and in a self-responsible manner: They elaborate on relevant topics and spread related knowledge to the cross-functional managers. These managers are developers from each team that have—in addition to their weekly development tasks—the responsibility to share the knowledge they acquired during the cross-functional team meetings with their fellow development team members.

Most of the cross-functional work is done at the beginning of the course, however, meetings are conducted until the end of the projects to ensure a high quality. This approach allows the course organizers to distribute knowledge from instructors to the cross-functional coaches, followed by an information sharing via the cross-functional managers to all students of the course. Major challenges are timing and coordination between project and cross-functional teams, as well as with course-wide events, such as the course-wide lectures, or presentations in which the project teams present their status to the whole course and all customers.

So far, the iPraktikum has seen multiple cross-functional teams, such as a modeling management team to support managers in creating, reviewing, and refactoring software models [11] or a release & merge management team to support managers in setting up the infrastructure, i. e., continuous integration and delivery, as well as ensuring the branching model and the code review process [177]. With the **Cuu** syllabus, we extend the iPraktikum by a dedicated **Cuu**$^{\textbf{SE}}$ management cross-functional team, with a **Cuu**$^{\textbf{SE}}$ instructor, **Cuu**$^{\textbf{SE}}$ coach, and **Cuu**$^{\textbf{SE}}$ manager.

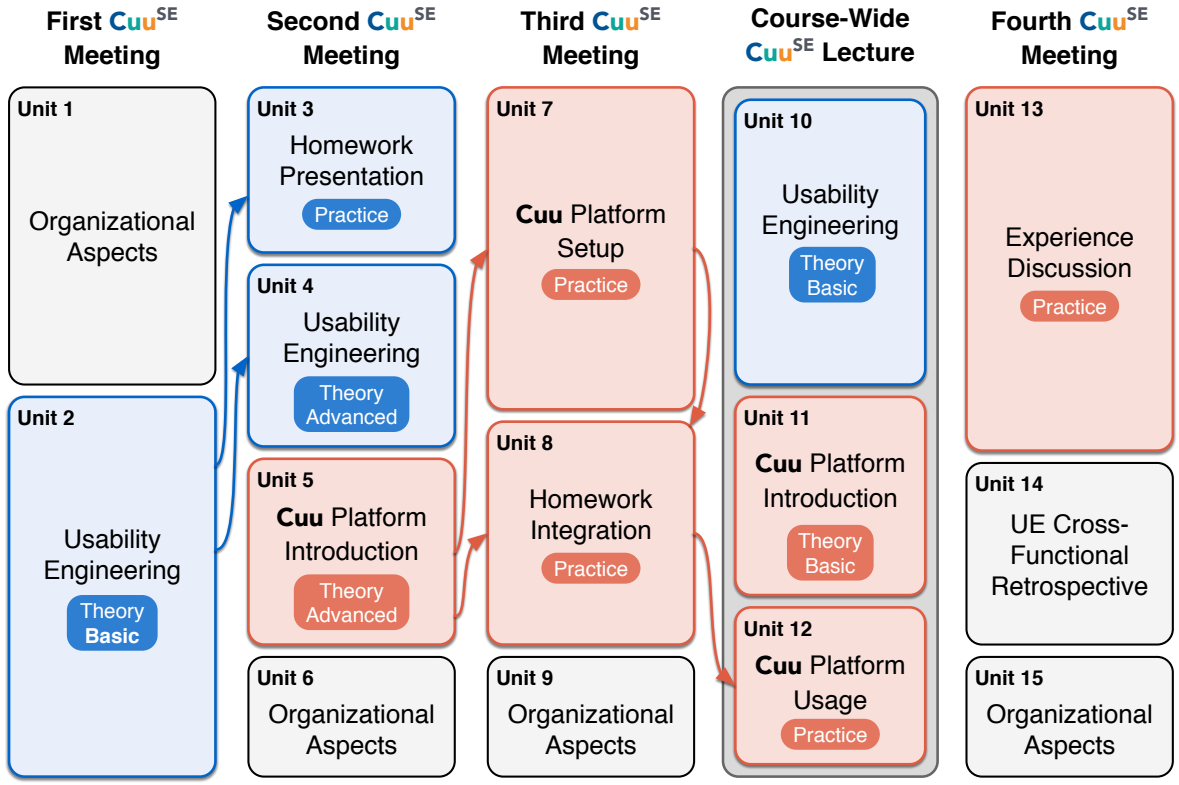


Figure 11.3: Overview of the **Cuu** syllabus, adapted from Johanssen *et al.* [152].

11.3 Teaching Continuous User Understanding

We introduce the **Cuu** syllabus as a guideline for teaching usability engineering in cross-functional teams within multi-project courses. In Figure 11.3, a visual overview of the **Cuu** syllabus shows the schedule for teaching continuous user understanding to introduce both **Cuu**^{SE} and usability engineering in cross-functional teams. Each column represents a full meeting, which lasts for approximately 90 minutes; the time frame of the course-wide lecture might be shorter. Every box represents a **Unit**, which encapsulates similar content and is presented as one block within a meeting. The order of units suggests their actual position within the meeting, while the size correlates with the time required for the unit. Blue units address usability engineering content (**Unit 2, 3, 4,** and **10**), while orange units (**Unit 5, 7, 8, 11, 12,** and **13**) address the **Cuu** platform. Grey units related to organizational aspects of the course (**Unit 1, 6, 9, 14,** and **15**). Colored arrows indicate constraints between units: for instance, **Unit 2** must precede **Unit 3** and **Unit 4**, while **Unit 10** can be introduced to students at any time. The *Theory* and *Practice* tags distinguish the theory units from hands-on units; theory units vary in their level of difficulty: *Basic* content includes overviews and topic introductions, while *Advanced* content extends basic unit knowledge.

The syllabus aims to stepwise introduce relevant concepts and to reduce theory

parts in favor of practical elements over time. A course-wide lecture, which all students of the course attend, serves as a common point of reference and milestone for the CuuSE managers. More details about this lecture is provided in Section 11.4.2.1, while in the following, we detail every meeting with its content and goals.

11.3.1 First Meeting

The focus of the first CuuSE meeting lies on the theory of usability engineering, which we base on existing and fundamental work [216, 227]. The meeting should clear up any organizational questions the CuuSE managers might have at the beginning of the course.

The first CuuSE meeting consists of many organizational aspects (**Unit 1**) and starts with an introduction round of all CuuSE managers. They are asked to include a brief overview of the projects they are working on. This should prepare them to be able to provide feedback to any of the developed applications during a later point in time of the project. Hereafter, we introduce them to the responsibilities of the role being a CuuSE manager. We summarize them as follows:

- Learn, repeat, and consolidate the general concepts of usability engineering.
- Integrate the Cuu framework for user understanding in the teams' project and promote its utilization during the development.
- Drive the realization, i. e., the implementation, of new insights that you will gain from usage knowledge and usability testing.

As the major element of the first CuuSE meeting, the CuuSE coaches prepare and hold a presentation regarding basic usability engineering knowledge. This is based on the book *Usability Engineering* by Jakob Nielsen [216]. The coaches are asked to focus the presentation on the following key aspects:

- overview of usability slogans;
- introduction to the usability engineering lifecycle;
- different forms of prototyping as well as their benefits and challenges, including both tools and best practices;
- overview of usability heuristics which should prepare the students for working on the homework for the following meeting;
- how to use the Thinking Aloud protocol as an example for usability testing;
- key idea of discount usability engineering.

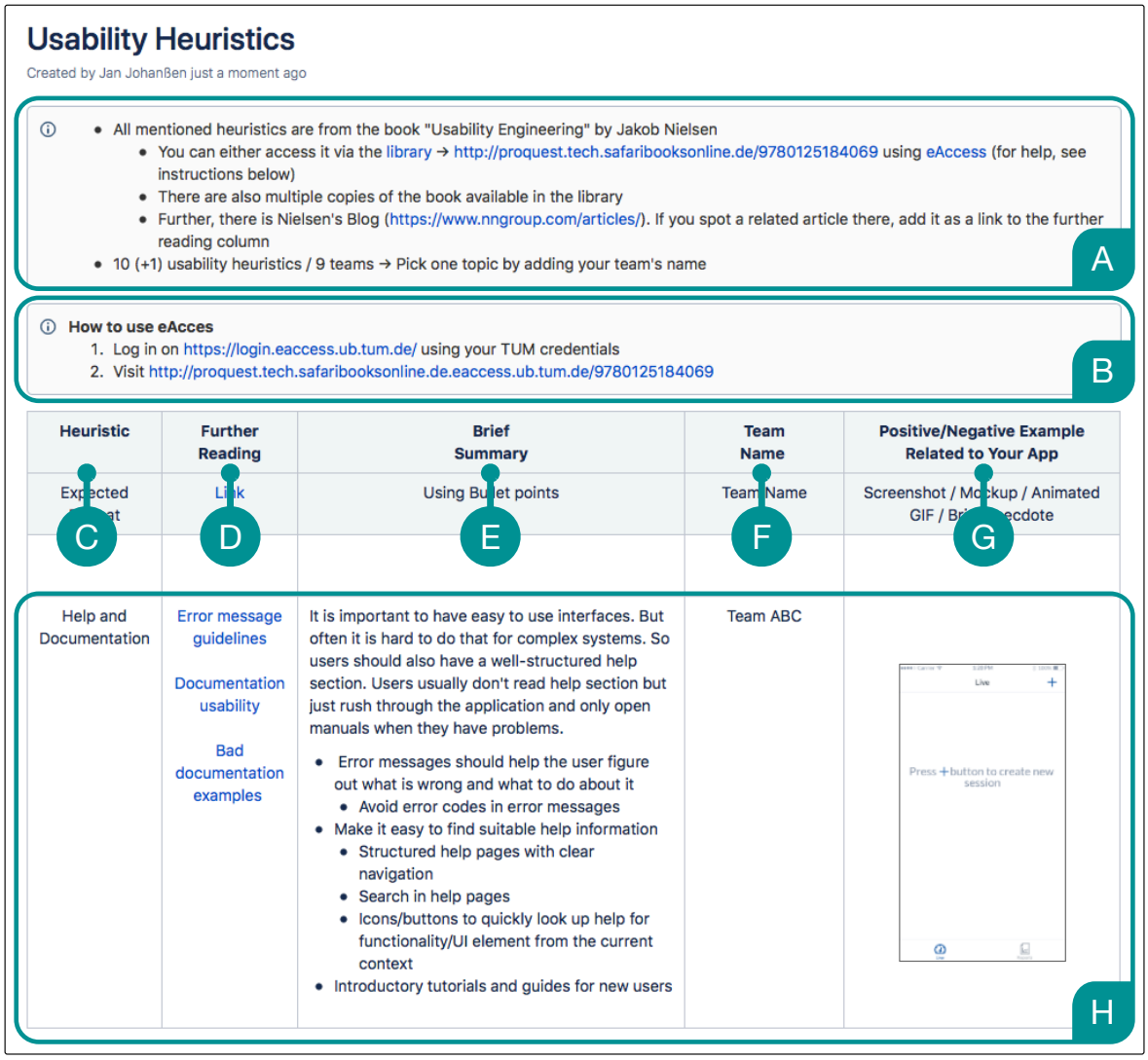


Figure 11.4: The first homework page, adapted from Johanssen *et al.* [152].

The presentation should be interactive and the coaches are encouraged to invite the managers to contribute ideas and descriptions while presenting the slides. Often, the managers already have a common understanding of the concepts, however, they cannot formally refer to it (e. g., the Thinking Aloud Protocol).

The homework is introduced at the end of the meeting; it reflects a first step of applying theoretical concepts to the team's applications. We ask the managers to pick and research one usability heuristic and share their results within the second **Cuu**$^{\mathbf{SE}}$ meeting as part of a two-minute talk. In Figure 11.4, we show a wiki page prepared for that purpose; each manager has to fill in one row. The page contains the following parts: Ⓐ Description of homework. Ⓑ Help on how to access supportive material. Ⓒ Heuristic name. Ⓓ Link to further reading. Ⓔ Brief summary of the heuristic. Ⓕ The team's name. Ⓖ Concrete example of the heuristic within the team's application. Ⓗ An instance for an entry provided by the teams.

We are particularly interested in the CuuSE managers' results on (Ⓖ), the transformation of the usability heuristic to their application. We ask them to create mockups of good and bad examples of the heuristics. Therefore, it is important to ensure that the CuuSE managers have access to the book, either by lending a copy from the university library or having access to an online version. Therefore, we enriched the wiki page, on which CuuSE managers were asked to add their homework, with a brief introduction on how to access the online library. The homework fits to the size of the iPraktikum (Section 11.2) with 10 teams and 10 usability heuristics.

The meeting closes with more organizational aspects, i. e., discussing meeting times, outlining the upcoming meetings, and clarifying questions. Before and after the first CuuSE meeting, the coaches are asked to publish additional content that is relevant for the managers, such as information pages about prototyping and links to popular tools. Not only do the coaches create and maintain these pages, they actively inform the CuuSE managers about the availability through a chat messaging platform. The coaches also ensure that the CuuSE managers work on their homework; this is supported by creating issues within an issue tracking system. This should prevent CuuSE managers from forgetting their homework, which would reduce the learning effect for all the others during the second CuuSE meeting.

11.3.2 Second Meeting

The second CuuSE meeting starts with a short presentation of the usability heuristics homework by every CuuSE manager (**Unit 3**). During the presentation of this first homework, the CuuSE instructor facilitates the presentations of the CuuSE managers. This is based on the assumption that the usability instructor has a broader perspective on the topic and can assess the heuristic. They know how to apply it and therefore can ask for typical problems and situations in which they occur, in case the CuuSE manager do not mention a relevant aspect as part of their summary. Furthermore, the CuuSE instructor helps to put the heuristic into context, e. g., by noticing relationships between them. Our goal is to encourage discussions about the topics and sharpen the CuuSE managers' senses and understanding for the topic.

As part of **Unit 4**, the CuuSE coaches hold a theory session on other usability engineering topics. This time, however, the information is shorter, approximately half of the time invested as in **Unit 2**, focusing on the following topics:

- small repetition of the topics of **Unit 2**;
- an introduction to scenarios as a usability engineering instrument;
- an open discussion on the definition of a feature;
- common practices, with minor focus on usability testing approaches different from Thinking Aloud and usability evaluation through heuristics.

The second and third aspects of these topics are the most important ones. Typically, the **Cuu**^{SE} managers are familiar with scenarios as a means to capture and describe requirements of a system. They also receive a repetition of this aspect in a proceeding course-wide lecture, which is not solely focused on usability engineering. However, in this unit, we address the usage of scenarios for usability engineering [217]. This prepares their understanding of feature crumbs, as it is required for usage monitoring Section 7.2. The open discussion intends to strengthen their understanding of features, how they can be represented, and to prove the point that the feature understanding is often a question of definition: They are defined in the way they are managed, e. g., user stories, scenarios, epics, and that they vary in size as well as other characteristics, such as the involved systems, stakeholders, criticality, or other aspects.

Unit 5 of the last unit of the second **Cuu**^{SE} cross-functional meeting is an introduction to the **Cuu** platform following these three aspects:

- Present the platform's functionality as described in Section 7.3.5 theoretically.
- Introduce the feature crumbs concept as described in Section 7.2, in particular walk the students through the creation of feature paths and the understanding of the feature status using Figure C.2.
- Explain the platform's goals and align them with the previously presented approaches for usability testing and evaluation. In particular, this includes an introduction of how feature crumbs can be used to perform *Heuristics Evaluation* [216], based on the knowledge that the **Cuu**^{SE} managers gained through the homework.

To be able to respond to the managers' questions about the platform and the crumbs, we advise the **Cuu**^{SE} coaches to carefully inform themselves using the relevant literature [154, 155]. We also provide a short and informal summary of both literature resources in form of a wiki page to ensure knowledge exchange.

We close the meeting with **Unit 6**, organizational aspects, which, for the most part, introduces the homework: The **Cuu**^{SE} managers are asked to prepare a feature path of a specific scenario in their application as a preparation for the third **Cuu**^{SE} meeting. In order to provide guidance, we provide a wiki page, as depicted in Figure 11.5, which we ask every **Cuu**^{SE} manager to copy into their team's space and provide a link for reference in a shared table. We modified the wiki page created for the feature crumbs homework for visualization: i. e., the description of text elements has been removed. Ⓐ requires a table of crumbs to describe the **Cuu**^{SE} managers' rationale when designing the crumbs. Ⓑ shows an example of how to add the crumbs on a code level. Ⓒ requires a JavaScript Object Notation (JSON) file that allows the manager to formally specify the feature path.

Feature Crumbs Templates

Created by Jan Johanßen, last modified 10 minutes ago

1. Define a flow of events on the Confluence page

Description text 1

Scenario Name	<Scenario1 Name>	Description	Scope in Code
Flow of Events	<Crumb1 Name>	The user taps on the "try feature" button	Triggered by an IBAction linked to the button
	<Crumb2 Name>	The user reads the presented article	Triggered by the UIScrollview and recognized by the UIScrollviewDelegate's *scrollViewDidScroll*
	<CrumbX Name>	...	... (Be creative on what interactions could be used to trigger Feature Crumbs! 🙂)

A

2. Add feature crumbs as templates to your project

Description text 2

UIViewController triggering Crumb Tracking

```
class ViewController: UIViewController {

    @IBAction func didTapTryFeatureKitButton(_ sender: UIButton) {

        print("Your Application Logic")
        // Feature Crumb Name: "<Crumb1 Name>"

    }
}
```

B

3. Prepare JSON snippet on the Confluence page with the flow of events

Description text 3

JSON Feature Representation

```
[
    {
        "step":1,
        "crumbName":"<Crumb1 Name>",
        "crumbType":"action"
    }
]
```

C

Figure 11.5: The second homework page, adapted from Johanssen *et al.* [152].

The managers start by creating a table that is similar to a formal representation of a scenario as shown in Figure 11.5 (Ⓐ). Each row of the tables describes a feature crumb, split by its name, a short description and the rationale of the crumb, and under which circumstances the crumb is triggered. Then, we ask the managers to add comments to their source code where they think the individual crumbs from the table should be triggered. We use the comments as a first step toward the actual tracking of features. This *dry run* requires them to put a focus on designing the feature, rather than starting with the tool that is able to observe a feature. We provide an example of how we expect this comment to look like in Figure 11.5 (Ⓑ); we put an emphasize on the naming, since it must follow the exact same spelling as in Figure 11.5 (Ⓐ). This should highlight the requirement that the same name of one row needs to be exactly spelled as in the code. The same holds true for the JSON file that needs to be provided by the managers in Figure 11.5 (Ⓒ). Here, they design a machine-readable version of the feature representation that they previously described within the table in Ⓐ. As with the comment-styled crumbs, this prepares an easy transition into using the tool in the next meeting.

11.3.3 Third Meeting

The third **Cuu^{SE}** meeting consists of primarily hands-on units. This enables each team to utilize the platform independently. In **Unit 7**, we provide a stepwise guidance on how to setup the **Cuu** platform and integrate the required components for an example feature, followed by the integration of their first own application-related feature in **Unit 8** that they prepared as part of their homework from the previous meeting. As a result, **Unit 7** forms the basis for **Unit 8**.

As this meeting relies on results from multiple units, the synchronization becomes a major challenge when preparing this meeting. On the one hand, the teams require a code basis which can be released and run on a device, which puts out requirements for both, the code and the workflow for its processing. On the other hand, in practical terms, the **Cuu^{SE}** managers require the hardware, i. e., computer and a mobile device, to run the steps that we describe in the following to setup and use the **Cuu** platform.

For **Unit 7**, we provide seven wiki pages that describe everything required to set up the platform and make use of it. As they contain potential pitfalls, we walk the **Cuu^{SE}** managers through them step by step, as described in the following.

Step 1: Login and Navigate to Project

As the **Cuu^{SE}** managers have never used the platform before, it needs to be ensured that they know where to find the platform and how to access it.

Step 2: Define new Feature and link Feature Branch

This step ensures that the **Cuu**^{SE} managers adhere to the iPraktikum's development process: All features and their respective branches are based on issues. We show this procedure to formally create a branch. Then, the managers switch to their **Cuu** project and create a feature in the platform, as depicted in Figure 7.16.

Step 3: Initialize SDK in Code Project

The **Cuu** platform requires the **Cuu** SDK (Section 8.1.2) to observe the activation of the feature crumbs and retrieve usage knowledge from **Cuu** kits. In this step, the managers integrate the SDK into their projects. They need to register the SDK with the *Tracking Token* and *Project Name* they extract from the services screen (Figure 7.19) to ensure that the **Cuu** SDK can push information to the **Cuu** platform.

Step 4: Add Feature Crumbs to Code Project

In this step, we introduce the **Cuu**^{SE} managers to the notation of triggering a feature crumb. At the bottom line, this is a method call with the feature crumb name as a string parameter, as described in Section 8.1.2.1. We ask them to seed a feature crumb called *My First Crumb* in the startup sequence of the application. Hereafter, the managers need to push all the latest changes to their remote repository.

Step 5: Add Commits to Branch

The new feature increment needs to be registered in the **Cuu** platform. The managers copy and paste the latest commit hash to their feature in the **Cuu** platform.

Step 6: Add new Feature Path

To associate a commit with a feature path, they select the commit in the **Cuu** platform and use a popup window to add a feature path in the JSON format. We ask them to use a single-step path, with step information *1* and crumb name *My First Crumb*.

After the managers performed these six steps, the feature is ready for usability assessment.[33] To test if the setup was correct, they need to release the latest commit. We introduce them to the different widgets and how they show a feature's performance. By asking the managers to add only one crumb with the same name and at the same position, we eliminate confounding factors that distract the **Cuu**^{SE} managers from their actual goal: setting up the feature for tracking. After they followed all steps and observed the widgets, we can be sure that everything is set up.

[33]Note that *Step 5* was required as the syllabus was validated with an alpha version of the **Cuu** platform, which did not allow automatic commit discovery via hooks. In addition, *Step 6* relied on a JSON file, which was replaced in the beta version of **Cuu** platform as shown in Figure 7.17.

To bring their focus back to the projects, we return to their prepared tasks from the homework in **Unit 8**. This is more complex than the example used for setup, but of actual relevance for the **Cuu**$^{\textbf{SE}}$ managers. As a side effect, they get to know the workflow of the **Cuu** platform even better. The **Cuu**$^{\textbf{SE}}$ managers learn how to add new crumbs to a feature path and what this means for the analysis, i. e., that it results in new feature version, by replacing the prepared crumb comments with actual method calls.

At the end of this meeting, every team has created at least one feature that can be observed using the **Cuu** platform. This result serves as the input for a hands-on exercise in the course-wide lecture as part of **Unit 12**, making **Cuu**$^{\textbf{SE}}$ useful for all students.

The meeting is closed by an additional round of organizational aspects as part of **Unit 9**. In particular, we explain to the managers how they can invite their team members to the project using a user management screen that is part of the **Cuu** platform, which we depicted in Figure 7.18. As this is an important requirement for the course-wide lecture, because otherwise the fellow team members will not be able to use the **Cuu** platform and the newly setup feature. Therefore, we track the progress of inviting all team members in order to ensure the effectiveness of the lecture. Eventually, we encourage the managers to continue using feature crumbs from this meeting on to collect and note down any observations, feedback, and lessons-learned, as a preparation of insights which could be shared during the fourth **Cuu**$^{\textbf{SE}}$ meeting.

11.3.4 Fourth Meeting

The fourth **Cuu**$^{\textbf{SE}}$ meeting serves as an opportunity to stimulate an inter-project discussion between all **Cuu**$^{\textbf{SE}}$ managers. While the goals of this meeting are manifold, the focus lies on **Unit 13**, in which every **Cuu**$^{\textbf{SE}}$ manager presents a feature that they set up in **Cuu**$^{\textbf{SE}}$ and observed over a longer time frame in more depth. This also includes the presentation of the created feature paths. As this fourth **Cuu**$^{\textbf{SE}}$ meeting is set at a later point of the project, we expect that the **Cuu**$^{\textbf{SE}}$ managers not only present the feature composition, but also report on how the observations helped them in terms of usability evaluation and assessment, as well as user understanding. Ideally, the managers elaborate on usage knowledge gained from the feature crumbs observations and how it supported them to solve a problem at hand. These in-depth descriptions of actual insights in real applications can also be beneficial for the **Cuu**$^{\textbf{SE}}$ managers of other teams, as they might face similar challenges.

Unit 14 is a retrospective in which the managers provide feedback on the work with the **Cuu** platform and the overall composition of the **Cuu**$^{\textbf{SE}}$ cross-functional role, as well as the **Cuu** syllabus. **Unit 15** closes the meeting with a general information regarding the remaining semester, such as the managers' responsibilities for future deliverables or how they can approach the instructors if questions remain.

11.4 Validation of the Cuu Syllabus

We introduced the **Cuu** syllabus as an instrument to disseminate **Cuu^{SE}** to student developers and to address the Instrumental Design Problem. This section is concerned with the validation of the syllabus within an instance of the iPraktikum.

11.4.1 Quasi-Experiment Design

We describe the design of the quasi-experiment by refining the high-level knowledge question into research questions; hereafter, we present the research method.

Refinement of Knowledge Question

The validation of the **Cuu** syllabus is a first step to answer Knowledge Question 5. This is based on the assumption that the success of the syllabus as an instrument to disseminate the **Cuu^{SE}** framework depends on the validity thereof, i. e., whether it supports developers in establishing a continuous user understanding.

To investigate the validity of the syllabus, we developed three RQs. First, we want to investigate its effects on the environment in which it will be applied; this addresses organizational aspects regarding the numbers of weeks that are needed for an instantiation, number of coaches that are required to implement the syllabus, as well as the number of meetings that are required. We summarize this as follows.

> **Research Question 17:** How does the **Cuu** syllabus affect the organizational aspects of the target environment?

The main goal of the syllabus is to disseminate the **Cuu^{SE}** framework to developers. We use their responses toward the homework of the syllabus, i. e., the degree to which they were able to create feature representations, as a proxy measurement to quantify the syllabus' support. We summarize this as follows.

> **Research Question 18:** How does the **Cuu** syllabus support the developers in creating feature representation?

Finally, we were interested to which degree the syllabus motivates the developers to continue using the **Cuu^{SE}** framework after its meetings were performed; we measure this using quantitive metrics that we derived from the platform usage. We summarize this as follows.

> **Research Question 19:** How dos the syllabus foster the usage of **Cuu^{SE}**?

Research Method

This section follows the process description of Wohlin *et al.* [314] to *"test the relationship between the treatment and the outcome"* [314], as described in Section 2.2.2.3.

Scoping We conducted a *Quasi-Experiment* [314] to study the performance of the **Cuu** syllabus. Our intention is to validate the **Cuu** syllabus as a treatment for **Cuu^{SE}** instrumentation. We study the effect of the syllabus on the dissemination of **Cuu^{SE}** in the problem context similar to the one described in Section 1.2. We report the experiment form the perspective of the **Cuu^{SE}** instructor as well as the researcher in charge of designing the **Cuu** syllabus.

Planning We applied the syllabus in an instance of the iPraktikum (Section 11.2) during the summer term of 2018. We hypothesize that the syllabus allows students to learn and understand the **Cuu^{SE}** framework in order to apply continuous user understanding. The choice of the iPraktikum sets the following *independent variables*: the number of project teams, the problem the projects try to solve, and the individual developers that become **Cuu^{SE}** managers. In total, we observed nine projects that address a heterogenous set of problem statements. The agreement to use the observation data was given on the basis of the managers' participation in the iPratkikum. We identified nine *dependent variables*, which describe measurements with respect to the organizational structure of the cross-project team, as well as to students' work with **Cuu^{SE}** during and after meetings—we detail them with Figure 11.6. We analyzed the treats to validity and detailed them in Section 11.4.3.2.

Operation We collected data for analyzing the performance of the experiment's variables from multiple source: First, data that affects the organizational aspects is derived from the course's organization and based on the instructor's decisions. Second, data that is produced as part of the homework is assessed by manual extraction from the respective wiki pages. Third, data that represents the usage of **Cuu^{SE}** is collected from the **Cuu** platform.

Analysis and Interpretation We describe the analysis of results in Section 11.4.2 and their interpretation in Section 11.4.3. While it offered all required functionality to support the syllabus, we consider the **Cuu** platform version that was used for the experiment as an alpha version. Therefore, besides the validation of the syllabus, we collected initial insights regarding the *perceived ease-of-use* [75] of the **Cuu** platform. For instance, the manual addition of commits as described in *Step 5* was perceived as inconvenient. This led to the integration of a hook system that automatically updates feature branches in the **Cuu** dashboard. In addition, some users reported that the registration of a feature path was cumbersome and error-prone. This was

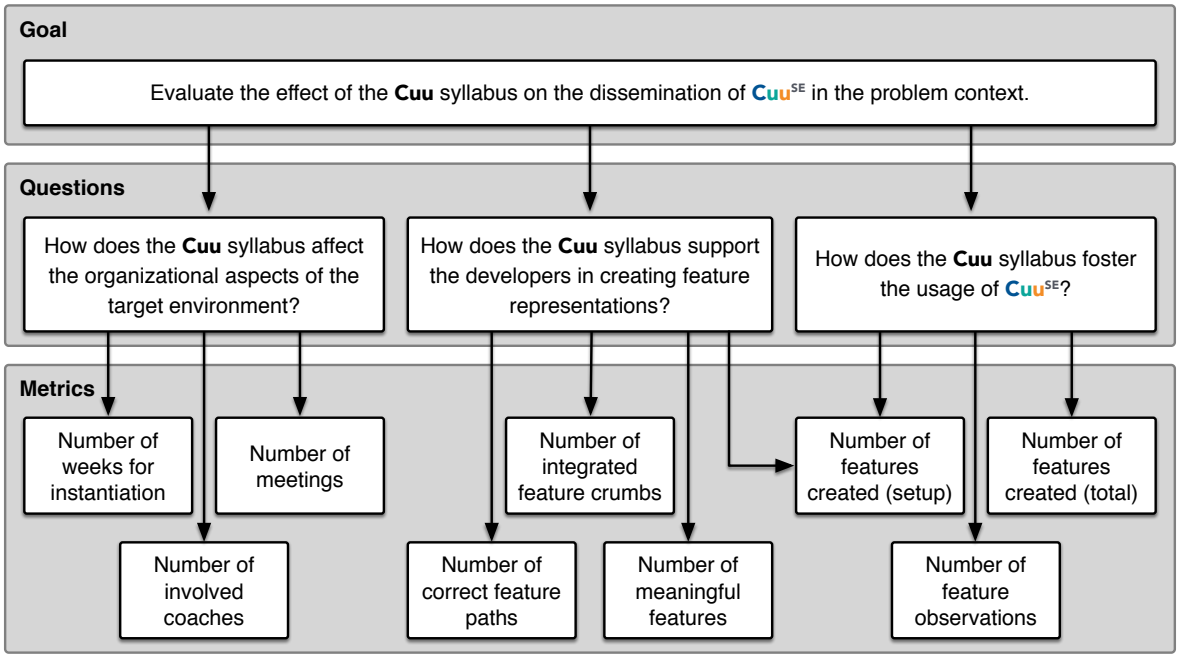


Figure 11.6: Visual overview of the validation of the **Cuu** syllabus following the GQM approach proposed by Basili *et al.* [26, 29].

due to the JSON format, which adds an additional manual step to the creation of a feature path. We addressed this with the addition of a drag-and-drop window. One user emphasized an increased effort to integrate crumbs into code. While the implementation of **Cuu**^{SE} does not provide a solution for this issue, we elaborate on future additions at the beginning of Chapter 13.

11.4.2 Results of Quasi-Experiment

Overall, we set out to ensure the applicability of the **Cuu** syllabus and in particular the acceptance of tool usage and theory concepts by the **Cuu**^{SE} managers. We use the GQM approach in order to provide quantitative measurements, all of which are objective metrics, as described in Section 2.2.3.1. For the purpose of the validation at hand and to comply with the GQM, we create the goal to *evaluate the effect of the **Cuu** syllabus on the dissemination of **Cuu**^{SE} in the problem context*. We summarize its relation to RQ 17, RQ 18, and RQ 19 in Figure 11.6. We describe the results for the three RQs in the following: organizational aspects (Section 11.4.2.1), feature creation (Section 11.4.2.2), and usage of **Cuu**^{SE} (Section 11.4.2.3).

Organizational Aspects

To make the **Cuu** syllabus transferrable to other courses, we tried to incorporate the units of the syllabus and their respective meetings into weeks of a typical semester of six month. In Figure 11.7, we depict the applied distribution of weeks and W1 is

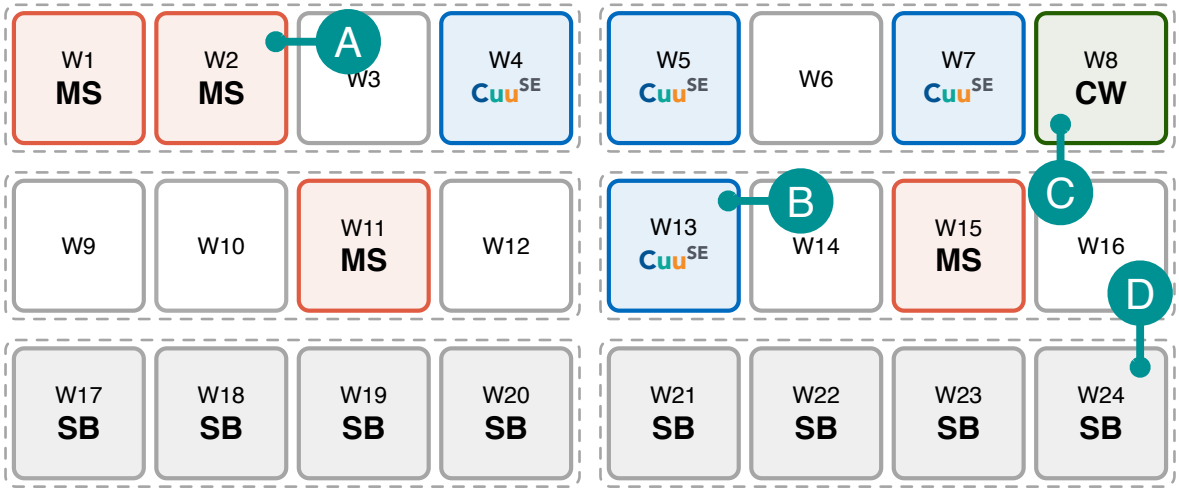


Figure 11.7: Chronological sequence of the iPraktikum in the summer term 2018, adapted from Johanssen *et al.* [152].

the first week of the semester, while W24 the last one. Supported by abbreviations, we use different box colors to map iPraktikum milestones, semester events, and **Cuu**$^{\textbf{SE}}$ meetings of the syllabus to semester weeks as follows: Red boxes indicate iPraktikum course milestones **MS** (Ⓐ). Blue boxes indicate **Cuu**$^{\textbf{SE}}$ meetings (Ⓑ). A green box highlights the usability engineering course-wide **CW** lecture (Ⓒ). Grey boxes relate to the semester break **SB** (Ⓓ), while the remaining weeks are solely used for development and other tasks.

With respect to the *Number of weeks for instantiation*, it requires a total of 13 weeks to fully implement the syllabus. Given the number of four meetings, this may appear high. However, there are many constraints as described in the following. As stated in Section 11.3, the syllabus builds upon the concept of cross-functional teams. These cross-functional roles require project teams and therefore can only be assigned after the initial iPraktikum student assignment to project teams has finished. As a result, the first two weeks of the semester *W1*, *W2* were reserved for the iPraktikum setup, while *W3* is concerned with getting the projects started and assigning the cross-functional managers within each project team. In *W4*, we ran the first **Cuu**$^{\textbf{SE}}$ meeting (Section 11.3.1), in which we focused on the organization, the usability engineering basics, as well as a first homework assignment. In the following week *W5*, the second **Cuu**$^{\textbf{SE}}$ meeting took place (Section 11.3.2), which encompassed the first practical exercises and the introduction to the platform. In the third **Cuu**$^{\textbf{SE}}$ meeting (Section 11.3.3) during *W7*, the tool support in form of the **Cuu** platform was integrated in each team's project by the **Cuu**$^{\textbf{SE}}$ managers, which also meant the first contact with the platform for each project.

After finishing the setup and explaining all concepts to the **Cuu**$^{\textbf{SE}}$ managers of each team, in *W8*, all course participants were introduced to the concepts of usability engineering and the **Cuu** platform as part of the course-wide lecture. The

course-wide lecture represents an optional addition to the **Cuu** syllabus; it provides an overview of the first three **Cuu^{SE}** meetings. As shown in Figure 11.3, **Unit 10** ensures a short theoretical introduction to set the stage. As part of **Unit 11**, we introduce the platform, how it can be used, and what it aims for. Finally, with **Unit 12**, we make use of the prepared features from the third **Cuu^{SE}** meeting to allow all students to benefit from a pre-defined feature within their own application to experience the usability assessment and evaluation methods of the platform.

The arrangement of the course-wide lecture left enough time to the teams to use the platform until the second milestone of the iPraktikum in *W11*. There, they were required to give an intermediate presentation of their work. In *W13*, two weeks after the intermediate presentation, but before the final presentation in *W15*, the fourth **Cuu^{SE}** meeting (Section 11.3.4) took place. We gathered feedback for the **Cuu^{SE}** cross-functional team and discussing feature paths and the corresponding feature crumbs.

With respect to *Number of meetings*, we were able to disseminate all 15 units within the planned four meetings, including the course-wide lecture. Based on a low number of experienced students, the *number of involved coaches* was one, however, they received strong support from the **Cuu^{SE}** instructors.

> **Summary RQ 17:** We were able to successfully integrate the **Cuu** syllabus into the iPraktikum as a target environment. While multiple constraints affect the number of weeks that are required to fully implement the syllabus, other organizational aspects are subject to the decisions of the instructors or the course organizers, which supports the flexible character of the **Cuu** syllabus.

Creation of Feature Representations

In this section, we report on the results from the homework of the **Cuu^{SE}** meetings, as they serve as an indicator whether the **Cuu^{SE}** managers understood the topic and whether they were able to successfully apply the presented concepts in practice.

The first homework that addresses usability heuristics was well perceived by the managers (Section 11.3.1). All of them were able to fill in the core components of the heuristic table. However, applying the heuristic to their actual application was only successfully performed by two teams; the other teams chose to use examples from well-known applications, such as Microsoft Word.

In the second homework, we asked the managers to define a feature that should then be used to observe feature crumbs and the platform (Section 11.3.2). We use these measurements to address the second question in Figure 11.6, i. e., the support in creating feature representations. We were able to derive more quantitative data shown in Table 11.1 to describe the respective metrics. In the table, every row depicts a feature; each team described at least one feature as part of their homework.

Table 11.1: Results from homework 2, adapted from Johanssen *et al.* [152].

Team	Number of Crumbs	Feature Path Correct?	Feature Meaningful?
1	3	Yes	Yes
2	12	Yes	Yes
3	3	Yes	No
4	14	No	No
5	3	Yes	Yes
6	3	Yes	Yes
	6	Yes	Yes
	6	Yes	Yes
7	3	Yes	Yes
8	3	Yes	Yes
	2	Yes	Yes
9	3	Yes	Yes

The *Number of integrated feature crumbs* in Table 11.1 shows that every team successfully defined feature crumbs for one of their application's features. At least two feature crumbs were used to describe the feature, while 14 was the maximum amount of feature crumbs used within the feature paths. On average, a feature path consisted of more than five crumbs. With respect to the metric *Number of correct feature paths*, except for one team, all other teams successfully described feature paths using the JSON format. However, as it can be inferred from the following section, team 4 was able to observe their feature at a later point in time, which suggests that they forget to define the feature path in the homework, but did so within the **Cuu** platform. With respect to the metric *Number of meaningful features*, out of the twelve features, we consider ten as a meaningful feature, i. e., it makes sense to be tracked and assessed for usability assessment. However, this was not the case for two features highlighted in red in Table 11.1: Team 3 implemented the feature path as three independent actions, while all started a new feature on their own. Team 4 described a massive feature path, which depicted almost all of the application's features. This led to a situation in which multiple consecutive features were described as one feature, which hinders their individual observation.

Summary RQ 18: Following the metrics that describe developers' performance during the second homework, we summarize that the syllabus supported the developers during the feature creation. This prepared them to set up **Cuu** features which can be observed for user understanding.

Table 11.2: Number of features created, adapted from Johanssen *et al.* [152].

Team	1	2	3	4	5	6	7	8	9
Features	3	3	1	1	2	2	2	3	3

Usage of CuuSE

We investigated whether the Cuu syllabus fosters the long-term usage of the CuuSE framework. Therefore, we collected more quantitative data, i. e., we counted the number of features that were created by the teams throughout the semester. Table 11.2 shows the results.

Comparing the numbers with the features that were created during the third CuuSE meeting, which can be derived from Table 11.1, we count 12 features for *Number of features created (setup)* and 20 features for *Number of features created (total)*, which makes eight additional features that were added over the course of the semester. Overall, the number of features created within Cuu fall short of our expectations. On average, every team created two features in Cuu, however, one of each of them originated from the third CuuSE meeting. While we do not have direct values for comparison, one indirect reference value might be the number of features the teams developed; based on this comparison, we would have expected between 5 and 15 features. We discuss this aspect in more detail in Section 11.4.3.1.

With respect to the *Number of feature observations*, we collected more quantitative data: Of the 20 features listed in Table 11.2, we recorded approximately 2200 feature observations as successful, while more than 850 were stopped with the possibility to be successfully finished, and more than 1500 were canceled. In total, more than 6500 feature crumbs were triggered. These numbers indicate that features observation was actively used by the Cuu managers.

Summary RQ 19: While the syllabus accomplishes its tasks of bringing students closer to the use of CuuSE and applying continuous user understanding, it runs short on lending weight to its effectiveness after the fourth meeting. Future versions of the syllabus can improve by enforcing the long-term usage of the CuuSE, in particular in case the developers require more support.

11.4.3 Discussion of Results

We distinguish the interpretation of the results into a discussion in Section 11.4.3.1 and threats to validity in Section 11.4.3.2.

Interpretation

This section discusses the results from the experiment, provides interpretations of the presented numbers, and points out challenges regarding various aspects of teaching continuous user understanding in cross-functional teams.

The Effect of Tool Support We summarize that the combination of the **Cuu** syllabus and the **Cuu** platform enabled the students to perform continuous user understanding in their real-world development setting.

At the same time, the quantitative analysis of the platform usage indicates that—after an initial phase—the teams reduced their efforts in designing feature paths and adding feature crumbs. We suppose that this may have various reasons; On the one hand, the **Cuu**$^{\text{SE}}$ managers have additional development tasks which may hinder them from performing usability engineering. On the other hand, we found a challenge in designing features, as described in the next paragraph.

The numbers suggest that we need to further support the teams in making use of the **Cuu** platform and point out the benefits to encourage them to invest more time in usability engineering. This effort should be addressed as part of future work.

Usability Engineering is Difficult. A closer look on the homework shows that the students struggle in applying usability concepts to their own applications. Regarding the first homework, the fact that only two teams were able to create applied examples of good and bad usability heuristic cases within their own application might be an indicator that these heuristics are difficult to understand, or to spot within their applications. The challenges in defining and understanding features become more obvious with the two instances that are highlighted as *"No"* in the column *"Feature Meaningful?"* of Table 11.1: The creation of a well-defined feature representation using feature crumbs requires effort, similar to test cases.

Notably, in case a team provided a name for their feature, they always correctly defined the respective feature path. This might provide an explanation why two teams created wrong feature path descriptions: They had a different mental model for the definition of a feature. This allows us to draw the conclusion that it is not enough to only provide tool support, but also concepts to work with it—which not strengthens the goal of aligning the syllabus next to the platform usage, but also increases the need for the **Cuu** workflow (Chapter 9).

Synchronizing Efforts Synchronizing the **Cuu** syllabus can become an issue. We identified the following aspects as particularly relevant:

- communicating the teaching materials and homework to project leads, coaches, **Cuu**$^{\text{SE}}$ managers, and the individual team members;

- aligning multiple meetings throughout the semester;
- ensuring that the teams' progress is mature enough to fit to the **Cuu** syllabus, in particular regarding the maturity of source code;
- connecting **Cuu**^{SE} managers with other cross-functional managers, such as the release and merge managers, to ensure that they have access to the repository when setting up **Cuu**, or are allowed to update third party repositories, which is a requirement to make the **Cuu** SDK work.

Enable Collaboration Besides the setup of an instant messaging channel for every project team, we also created a dedicated communication channel for the **Cuu**^{SE} cross-functional team. It turns out that such a channel can become an interesting and vivid place for discussions, which is important for usability engineering. Usability engineering is a collaborative process that requires multiple participants, both experts and users. In the channels, the students of other teams act as proxy users; while they only have a rough idea of the topic of the other applications, they know just the right amount of details to assess the usability of the application from the perspective of a user—even though they may not be the actual user audience.

In general, we can confirm the initially stated hypothesis by other researchers that usability engineering can be taught better with hands-on examples, which are shared with others to collect different opinions. An instant communication channel promotes the collaboration, while wiki pages—that we used for the homework to collect usability heuristics as described in Figure 11.4 as well as the feature creation in Figure 11.5—represent another point of interaction.

However, we noticed that many **Cuu**^{SE} managers still contacted the instructors directly in case of questions via a direct message and refrained from using the "public" channel to ask their questions to all the other students. We classified most of these questions as relevant for all managers. Therefore, we actively encouraged them to change this and will continue to work on this in future semesters.

Acknowledging Privacy Aspects of Users The understanding of users inherently relies on the interaction with users. With the **Cuu** platform and the **Cuu** SDK, the students obtain access to a tool that allows to learn more about the way a user interacts with an application. This includes—besides the ability to determine whether a feature has been started, completed, or canceled—additional knowledge sources that provide fine-grained information about the application usage and their users. Generally, **Cuu**^{SE} does not collect personal data that would allow for conclusions toward an individual user. Nevertheless, within the units of **Cuu** syllabus, we intend to sensitize the students for these aspects and increase their sense of responsibility as a consequence thereof. In particular, we ask the students to inform their end

users, i. e., the customers, about the addition of **Cuu** to their projects before they start using it. In addition, technical mechanisms, such as the start screen presented in Appendix C.1, inform the users about the data collection when they start the application and provide the ability to opt out.

Providing the Environment One additional challenge arises in providing the environment that is required to make use of the **Cuu** syllabus. Besides the operation of the **Cuu** platform and access to the **Cuu** SDK, to take full advantage of the syllabus, an extensive set of tools in form of infrastructure is required:

- an issue management system to enable tracking of tasks and development progress. During the experiment, we relied on Atlassian Jira[34];
- a wiki system to share information and enable a space to collaborate on the homework. During the experiment, we relied on Atlassian Confluence[35];
- a version control, continuous integration, and continuous deployment system that covers the full development process before the **Cuu** platform can be utilized. During the experiment, we relied on Atlassian Bitbucket[36] as well as Prototyper [9, 12];
- an instant messaging service. During the experiment, we relied on Slack[37].

Providing and maintaining this environment with the above-mentioned tool support represents a major challenge and is only feasible for large-scaled project courses.

Threats to Validity

This section briefly outlines the threats to the validity of the reported experiences centered around the four dimensions of validity [265], namely construct, internal, and external validity as well as the reliability.

Construct Validity Regarding the disparity between the intended and actual study observations, there might be the chance that the **Cuu** syllabus and its introduction to the iPraktikum did not contribute to the gain of knowledge for students with respect to continuous user understanding. However, as there has not been a dedicated focus of usability engineering in the course before, we can assume that any information regarding that topic is beneficial to the students. To ensure that the presented theoretical concepts are straightforward, easy-to-understand, and consistent,

[34] https://www.atlassian.com/de/software/jira
[35] https://www.atlassian.com/de/software/confluence
[36] https://www.atlassian.com/de/software/bitbucket
[37] https://slack.com/

we involved multiple instructors from other cross-functional teams to mediate the risk of ambiguous knowledge transfer. In addition, the **Cuu**^{SE} managers constantly reviewed the consistency of the provided documentation of the **Cuu** platform.

Internal Validity The correlation between investigated and other factors might become visible in the way the managers made use of the taught concepts. In particular other responsibilities, such as the coding task or other university courses, might have affected the extent to which the managers were investing effort in working as the role of a **Cuu**^{SE} manager. We tried to mediate this effect by reducing additional efforts, as well as providing help whenever needed.

External Validity The degree of generalizability of the study is low, as we can only report of one instantiation in the iPraktikum as the representation of an agile multi-project course. However, we argue that the presented syllabus is in general highly related to the structure of the iPraktikum, which makes generalization even more difficult. Still, we think that any observations and results help other lectures to improve their teaching efforts regarding usability engineering and generally support the introduction of **Cuu**^{SE} in project teams. To be able to report on more generalizable results, the repeated application of the **Cuu** syllabus is required to gain more reliable insights.

Reliability With respect to reliability, the fact that only one **Cuu**^{SE} instructor supervised the cross-functional team might have biased the application of the **Cuu** syllabus. We argue that this effect was rather low, as the cross-functional team was integrated in a wider course environment that was additionally supervised by three researchers that are part of the iPraktikum program management.

Chapter 12

Validation of the CuuSE Framework

> *"As one would expect from the hierarchical model of professional knowledge, research is institutionally separate from practice, connected to it by carefully defined relationships of exchange. Researchers are supposed to provide the basic and applied science from which to derive techniques for diagnosing and solving the problem of practice. Practitioners are supposed to furnish researchers with problems for study and with tests of the utility of research results."*
>
> — DONALD A. SCHÖN [280]

This chapter is concerned with a comprehensive validation of the joint utilization of multiple **Cuu** artifacts. We performed a technical action research which reflects an extension of the design science approach as described in Section 2.2.1.

We relied on a survey as an empirical strategy that we aligned with a client cycle to perform the technical action research. We chose the empirical strategy of a survey as *"they try to capture the results of having used a method or tool."* [245]. In Section 12.1, we provide details on its design: We created two questionnaires that we distributed to **Cuu**$^{\textbf{SE}}$ managers during an instance of the iPraktikum.

The focus of this chapter lies on the analysis of the questionnaire data. We base the validation on the Technology Acceptance Model (TAM, introduced in Section 2.2.3.2), which helps us understand the managers' perceived usefulness and their perceived ease of use, as well as their intend to use of the **Cuu**$^{\textbf{SE}}$ framework in future projects. We separate the results based on two questionnaires: While the first half of Section 12.2 describes the responses toward the general **Cuu** components, such as the **Cuu** workbench or feature crumbs, the second half is concerned with a more detailed analysis of individual widget visualizations and specific use cases of them. This initial validation of the **Cuu**$^{\textbf{SE}}$ framework serves as the basis for further improvement, before it can be fully evaluated within an industrial setting.

12.1 Survey Design

In this section, we describe six research questions that guided the design of the survey. Hereafter, we outline the research method, i. e., the conduction of a survey, that we followed to validate the **Cuu**^{SE} framework.

12.1.1 Refinement of Knowledge Question

With respect to Knowledge Question 5, we derived six RQs that address the managers' attitude in order to answer the question how **Cuu**^{SE} supports developers in establishing continuous user understanding.

We were interested whether the underlying assumptions that form the core for **Cuu**^{SE} are well-perceived by the **Cuu**^{SE} managers. We summarize this as follows:

> **Research Question 20:** How do managers perceive the usefulness of general concepts that form the basis for **Cuu**^{SE}?

To make use of the **Cuu**^{SE} framework, the **Cuu**^{SE} managers need to initially define multiple parameters as part of the setup process. We intended to find out more about the managers' impressions regarding the mostly operational actions they performed while being guided by the **Cuu** syllabus. We summarize this as follows:

> **Research Question 21:** How do managers perceive the ease of use of the setup process of **Cuu**^{SE}?

A **Cuu** feature forms the basic entity within the **Cuu** platform. On their basis, the managers can inspect usage knowledge. We were interested in the managers' perception with respect to the feature management within **Cuu**^{SE}. We summarize this as follows:

> **Research Question 22:** How do managers perceive the ease of use of managing features?

After **Cuu**^{SE} is initialized and at least one feature is created, the managers can start analyzing usage knowledge for features. We were interested in the managers' work with the **Cuu** platform and in particular how they examined the dashboard. We summarize this as follows:

> **Research Question 23:** How do managers perceive the usefulness and ease of use of analyzing features in **Cuu**^{SE}?

The workflow to extract tacit knowledge from the usage knowledge is dependent on the usefulness of widgets as a means for visualizing the usage knowledge.

Therefore, we investigated the managers' perceived usefulness of core **Cuu** widgets that were presented within this dissertation, namely FeatureKit (Section 8.1.2.1), ThinkingAloudKit (Section 8.3.1), EmotionKit (Section 8.2.1), and PersonaKit (Section 8.5.1). We summarize this as follows:

> **Research Question 24:** How do managers perceive the usefulness of widgets visualizing usage knowledge from FeatureKit, ThinkingAloudKit, EmotionKit, and PersonaKit?

Finally, based on their previous, hands-on experience with **Cuu^{SE}** over the period of a semester, as well as potential use cases that were demonstrated using a visionary scenario—as described during the preparation phase of the survey—we were interested in the managers' intention to use **Cuu^{SE}** in future projects of theirs. We summarize this as follows:

> **Research Question 25:** Do managers intend to continue using **Cuu^{SE}**?

12.1.2 Research Method

We summarize the activities required to conduct a survey based on the three phases that we introduced in Section 2.2.2.2.

Preparation

The goal of the survey is to answer Knowledge Question 5 that we derived from the Knowledge Goal 2 that is concerned with the validation of the **Cuu^{SE}** framework. The survey is set in the environment of the iPraktikum, as described by in Section 11.2. Multiple actors were involved: The **Cuu^{SE}** instructor, who also represents the researcher conducting and administrating the survey; the **Cuu^{SE}** coaches, who supported the instructor in their activities; the **Cuu^{SE}** managers, as the recipients of the survey's questionnaires, i. e., the subjects to be surveyed.

We prepared the questionnaires and carefully formulated questions that are based on examples provided by Davis *et al.* [74, 76]. However, we made adaptions to the way we phrased the questions, in order to reflect the academic environment in which the managers, i. e., the students, were working. For instance, this meant to omit the addition such as *"at my job"* during questions addressing the usefulness. Nevertheless, we asked the managers to always make sure that they should answer the questions from the viewpoint of their role as a **Cuu^{SE}** manager.

We used Likert-typed questions with a 5-point scale [194] considering that it provides almost similar results as comparable 7-point or 10-point scales [77]. In addition, we used free text questions in order to stimulate more verbose responses that provide the underlying rationale for the qualitative Likert questions. Overall, we

prepared 23 Likert-typed questions that we supported with 27 additional free text questions. In case a question required a more elaborated explanation, we added a descriptive text or screenshot to support its understanding. For the second questionnaire, we created an example scenario that should support most of its questions to provide a vivid reference when providing qualitative responses in the free text forms. We designed the individual questionnaires to not take longer than 15 minutes to be filled in by the managers. Both questionnaires are depicted in the Appendix D.4 and Appendix D.5.[38]

We relied on a beta version of the **Cuu** platform that included improvements from the validation process described in Chapter 11. As surveys rely on a standardized format [204], we limited the scope of the answers to prevent extensive responses, which appeared appropriate for the maturity level of the **Cuu** platform.

Following the categorization of surveys in Section 2.2.2.2, we classify the survey as both explanatory and explorative: While its first questionnaire emphasizes the perceived ease of use, the second questionnaire focuses on the validation in terms of perceived usefulness and the managers' intention to use the framework. Based on these results, we can understand the managers' motivations and use these insights to work on a new version of **Cuu**$^{\textbf{SE}}$ that can be evaluated in practice.

Conduction

We conducted the survey during the winter semester 2018/19 of the multi-project course iPraktikum at the Technical University of Munich. The course encompassed a continuous user understanding cross-functional team that was instructed based on the **Cuu** syllabus, as introduced in Chapter 11.

The managers were individually elected within the teams; overall, 10 managers participated in both questionnaires. Their agreement of using the collected data was given on the basis of their participation in the iPratkikum. In addition to the managers, three **Cuu**$^{\textbf{SE}}$ coaches were part of the cross-functional team and in charge of administrating the survey process, i. e., distributing both paper-based and online versions of the questionnaires, ensuring their completion, as well as providing help in case questions were raised.

Utilization

We use TAM as described in Section 2.2.3.2 to analyze the survey data. As TAM is also adequate for validating technologies that are in an early stage of development [76], its applicability fits the scope of the **Cuu**$^{\textbf{SE}}$ framework. The reminder of this chapter is concerned with reporting the results.

[38] Note that we omitted one page containing widgets that are not addressed in this dissertation.

12.2 Results of Survey

In this section, we present the results on the CUU^{SE} managers' responses toward general usability engineering concepts, setting up and working with CUU^{SE}, as well as their overall understanding. This concerns the general appearance of the workbench, as well as their understanding of working with features in CUU^{SE}.

12.2.1 Concepts for User Understanding

Following RQ 20, we investigated the managers' perceived usefulness of general concepts that form the basis for CUU^{SE}. Table 12.1 depicts their responses.

Table 12.1: Managers' perceived usefulness of concepts for user understanding.

ID	**Question** (see Figure D.4)	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Q1.1	A usability engineering platform is useful to establish a common understanding of a feature.	4	5	0	0	1
Q1.2	Feature branches are useful to collect usage data for a feature.	2	5	3	0	0
Q1.3	The flow of events of a scenario is useful to analyze the usability of a related feature.	4	4	2	0	0

The responses regarding **Q1.1** are diverse and address platforms for usability engineering. Generally, with one exception, all managers agreed or strongly agreed that a usability engineering platform is useful to establish a common understanding of a feature. Four managers highlighted that it helps to show the degree to which users understand the concept of a feature and how they make use of it; in particular with a focus on whether the flow is clear to them. Generally, they argue that any information that can be collected to improve a feature is valuable. Three other managers described the use of such a platform in a way that it helps them to detect situations in which the users' behavior derives from the expected behavior. It supports them in detecting issues that block users from continuing the use of a feature. Another two managers elaborated on the concept of such a platform; they supported the idea that the feature definition is driven by the use of branches, which is why the feature's scope should not exceed a certain size. In addition, they note that they would expect that such a platform is able to support the comparison of multiple features, motivated by the goal to see whether on feature starts replacing

another one. One manager strongly disagrees with the question and argues that their application's architecture does not fit the way **Cuu^{SE}** works.[39]

With respect to the use of feature branches as a means to collect usage data for a feature **Q1.2**, no manager disagreed; one manager did not provide a qualitative answer. Two groups of answers can be identified. Five managers highlighted the benefit of feature branches to focus their attention on one particular aspect of a feature. They stated to welcome the separation and fine-grained analysis of user feedback on the basis of individual features; without interfering with other features that might create a confounding variable. They noted that it is in particular useful in case a feature consists of complex structures. On the other hand, while they generally appreciate the format of feature branches, four managers expressed the need to release more than one feature at a time. Reasons therefore lay in the fact that they are either not releasing feature branches to users, or that they think that small-sized features would be tedious to manage.

In **Q1.3**, we asked the managers for their opinion on the usefulness of events of a scenario in order to analyze the respective feature. All managers either agreed or provided a neutral answer. Seven managers highlighted the ability to extract insights on how the users uses a feature. They noted that it provided structure during the analysis process and enables the correlation to statistics. Three managers, who responded with two neutral and one agreeing statement, made the usefulness dependent on the type of scenario, i. e., whether it depicts a linear flow or not. We detail this aspect as part of **Q4.2** in Section 12.2.4.

> **Summary RQ 20:** Managers predominantly agree that a dedicated platform for addressing user understanding activities benefits their work. They mostly agree that feature branches are a suitable means to collect usage data, given that they can be easily managed. Analyzing usage data on the basis of scenario events is perceived as useful by managers.

12.2.2 Setup Process

Following RQ 21, we investigated the managers' perceived ease of use regarding the setup of **Cuu** platform. Table 12.2 depicts the managers' responses.

One of the main goals of the **Cuu** syllabus is to ensure that managers are provided with all required information to make use of **Cuu^{SE}**. We validate this aspect using **Q2.1**. Eight managers either agreed or strongly agreed that enough information was provided. They emphasized that in particular the wiki pages are easy to

[39] Note that based on the response the respective manager provided in this and multiple other questions, we assume that this manager did not fully understand some of the questions as we intended them. Nevertheless, we included all their responses in presentation for a thoroughly report of the results. We further address this aspect in the treats to validity (Section 12.3.2).

Table 12.2: Managers' perceived ease of use of the setup process.

ID	**Question** (see Figure D.5)	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Q2.1	Enough information to set up CuuSE was provided.	4	4	0	2	0
Q2.2	Linking CuuSE with Bitbucket is easy.	6	2	1	0	1
Q2.3	Initializing the CUU SDK within the Xcode project is easy.	4	6	0	0	0

follow; furthermore, they liked the step-by-step introduction in the **Cuu**$^{\text{SE}}$ meetings. One manager suggested to add a comparison with other similar platforms to provide more context. Two managers disagreed with the statement; while one manager criticized the lack of a confirmation step that indicates whether the **Cuu** platform is setup properly, another manager stated that they were unable to follow the instructions in retrospective, though they admitted that another team member attended the respective **Cuu**$^{\text{SE}}$ meeting.

In order to receive automatic code updates from a version control system, a configuration as described in Figure 7.19 is required. **Q2.2** addresses the managers' opinion whether this can be achieved without major efforts. Eight managers stated this is done *more than easy*, in particular following the provided step-by-step instructions. Two managers did not agree with the statement, while one of them highlighted that the management of permissions was difficult; this was needed as the **Cuu**$^{\text{SE}}$ managers generally do not have access to the configuration page of the version control system. Additional communication with other cross-functional teams was required, a challenge that we already identified in the validation of the syllabus.

As we described in Section 8.1.2, the managers are required to perform multiple steps to initialize the **Cuu** SDK within their source code projects. With respect to the answers to **Q2.3**, all managers spoke with one voice in agreeing that the initialization process is easy—mostly justifying that it requires only a couple of one-line statements. One manager note that they were temporarily struggling to update to the latest version of the SDK, which was related to another issue and which they were able to resolve.

Summary RQ 21: With two exceptions that provided concrete suggestions for improvement, all managers praised the extent of the documentation that was provided in order to setup **Cuu**$^{\text{SE}}$. The majority of managers (strongly) agrees that linking the **Cuu** platform with a version control system is effortless. Managers concordantly agree that the initialization of the **Cuu** SDK is easy.

12.2.3 Management of Features

Following RQ 22, we investigated managers' perceived ease of use of managing features in the **Cuu** platform. Table 12.3 depicts the managers' responses.

Table 12.3: Managers' perceived ease of use of the management of features.

ID	**Question** (see Figure D.6)	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
Q3.1	Creating a feature in the CuuSE platform is easy.	1	5	2	2	0
Q3.2	Adding feature crumbs in the source code is easy.	4	4	2	0	0
Q3.3	Creating a feature path in the CuuSE web interface is easy.	2	6	1	1	0

After a new feature has been created based on an issue, the creation of a feature in the **Cuu** platform can be initiated as depicted in Figure 7.16. With **Q3.1**, we asked the managers on their opinion on this process. Six managers agreed that it is easy, noting that it aligns well with the standard procedures of creating a feature in other platforms such as version control systems or the issue tracking system. At the same time they stated that the close connection to other systems comes at the cost of dependencies, such as when the systems are not available, or the inability to delete features in the **Cuu** platform as they would still be available in the external system. Two managers took a neutral stance, which was related to technical issues in the user interface of **Cuu** platform. Two other managers disagreed with the statement. One explained that the process is non-intuitive and tedious, while the other manager noted that they are unable to reflect a full feature within **Cuu**$^{\text{SE}}$ as it consisted of two different feature branches.

The management of features includes the addition of feature crumbs in the application's source code as described in Section 8.1.2.1. With **Q3.2**, we investigated managers' thoughts on whether this can be considered an easy process. With two neutral responses, the majority of managers agreed that the addition of feature crumbs to source code *"couldn't be simpler"*. They appreciated the lightweight character that is *"just like a print statement"*, while they also acknowledged that the repeated addition of feature crumbs means a lot of work. Two managers remarked the challenge in finding the best place to add the feature crumbs; they emphasized that it is hard to find just the right place to obtain meaningful results.

Before usage knowledge for a feature can be inspected within the **Cuu** platform, the managers need to create a feature path as we described in Figure 7.17. Question **Q3.3** addresses the managers' opinion on this process. Eight managers agreed that the creation of a feature path is easy. They highlighted the simple and clean

user interface and the visual feedback after a feature path has been set. At the same time, some of them note that the slow processing and resulting waiting times sometimes reduces the overall user experience. Two managers provided critical responses, which may be attributed to extreme occurrences of such inconvenience: the platform's lag and down time hindered them in creating feature paths.

> **Summary RQ 22:** Most of the managers agreed that the creation of features is easy, however, improvements could be made with respect to usability. All managers affirmed the simplicity with which feature crumbs can be added to source code, while they acknowledged the challenge in finding the right place. The creation of feature paths within the **Cuu** platform is perceived as easy, however, technical issues lead to major restrictions for some of the managers.

12.2.4 Analysis of Features

Following RQ 23, we investigated the managers' perceived usefulness and ease of use regarding feature analysis in **Cuu**$^{\text{SE}}$. Table 12.4 depicts the managers' responses.

Table 12.4: Managers' perceived usefulness and perceived ease of use of feature analysis. For **Q4.3**, one manager did not provide a response.

ID	**Question** (see Figure D.7)	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Q4.1	Switching between the usage data for different features is easy.	3	3	3	1	0
Q4.2	A linear feature path is useful to describe our feature under development.	0	3	4	3	0
Q4.3	The relationship between the git graph springboard and the usage data widget in the CuuSE platform is easy to understand.	1	3	2	3	0

The **Cuu** platform focuses on the analysis of one feature at a time; however, it allows to switch between features as applications usually consist of multiple features. Using **Q4.1**, we validate the managers impression toward the functionality that we describe in-depth in Figure 7.15-Ⓓ. With six managers, more than half of the surveyed subjects agreed or strongly agreed that the selection process of a features is simple and intuitive, in particular given the distinction made by the orange color. The remaining managers took a reserved stance, with one of them disagreeing with the statement. All of them justified their answer with slow loading times of the feature content.

We introduced feature crumbs in Section 7.2.2 as a lightweight concept to monitor user interactions that relate to relevant feature particulars. We decided to focus on a linear feature path composition because with increasing complexity of feature structures, the analysis process would also increase with every additional usage knowledge source. On the basis of **Q4.2**, we strived to understand the managers' perceived usefulness of linear feature paths considering their applications.

Three managers noted that they agree with the statement and expected a linear sequence of feature crumbs; one of them noted that they have not worked with feature crumbs so far. The seven other managers provided critical answers. They stated that their features usually do not consist of linear feature paths and can have multiple starting points, while they understood the reason for **CuuSE** to build on linear feature paths observations and speak of a *"tradeoff [...] as it would be difficult to describe complex feature executions"*. They provided two examples of how they would imagine useful extensions. First, they would have appreciated the addition of loops that reflect smaller, repetitive parts of a feature that are performed multiple times or in a recursive way. Second, one manager addressed the ability to describe decisions, as parts of a feature may remain unused.

A core concept of the **Cuu** dashboard is the relationship between the springboard and the widgets as described in Section 7.3.2. Using **Q4.3**, we asked the managers about their opinion on whether this relationship is easy to understand; one manager did not provide a response to the Likert-typed question but provided a free-text response. We observed answers that are divided into two: Five managers supported the statement. They highlighted the visual clarity, described it as intuitive, and noted that released increments and their related usage knowledge can be easily identified. One of the five managers pointed out the information text that is shown when switching to either compare or range widgets, which instructs users to select a second commit. Another one of the five managers stated to prefer not to show widgets that do not yet provide usage knowledge, i. e., that only populated widgets should be presented to the user. In contrast, the second half of the managers struggled to understand the relationship between the two classes of dashboard elements; one of them noted that the focus on the code history is to prominent, while the widgets containing the important insights are rather hidden by placing them on the bottom of the dashboard. Minor aspects, such as adding a description of the used colors for the commits (see Figure 7.15-Ⓖ) was demanded by another manager.

Summary RQ 23: The managers agreed that switching between multiple features is intuitive, however, many perceived the waiting times as too long. The managers identified the linearity of feature paths as a weakness and provided suggestions for improvement. According to the managers, minor additions to the dashboard could noticeably improve the understanding of the dashboard.

12.2.5 Cuu Widgets

Following RQ 24, we investigated managers' perceived usefulness regarding multiple **Cuu** widgets. We omitted the report of answers to **Q2.3**, **Q3.3**, **Q4.3** (*"Provide a (visionary) example of how the {kit-name} widget and its results could be used to improve your application."*). Table 12.5 depicts the managers' responses.

Table 12.5: Managers' perceived usefulness of the **Cuu** widgets FeatureKit, ThinkingAloudKit, EmotionKit, and PersonaKit.

ID	Question (see Figure D.9 *et seq.*)	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Q1.1	Visualizing feature path observations separated by their state (Started, Completed, or Canceled) is useful to understand the feature's usability.	2	7	1	0	0
Q1.2	Visualizing the states of the feature path observations for unique device types is useful to understand the feature's usability.	3	4	3	0	0
Q1.3	Visualizing the number of feature crumb observations per individual feature crumb is useful to understand the feature's usability.	2	6	2	0	0
Q2.1	Visualizing Thinking Aloud data, i.e., the number of classified verbal feedback, in relationship to feature crumbs is useful to understand the feature's usability.	2	6	1	1	0
Q3.1	Visualizing the emotional response, i.e., the users' state such as happiness or anger, in relationship to feature crumbs is useful to understand the feature's usability.	3	2	2	2	1
Q4.1	Visualizing Runtime Personas for a release is useful to better fit the feature under development to users' requirements.	1	8	1	0	0

Every single one of the widgets was predominantly perceived positive by the managers. In total, summarizing the 60 possible responses, managers provided 46 instances of strongly agree and agree responses (76.67%), 10 instances with a neutral response (16.67%), and 4 instances in which they disagreed or strongly disagreed

(6.67%) with the usefulness of a widget. After the summary addressing the main feedback across all widgets, we detail responses per widgets in the following.

> **Summary RQ 24:** Most of the managers pointed out that the FeatureKit widgets are useful to initiate their analysis process and decide on whether to follow up on a potential issue. The managers predominantly welcomed the ThinkingAloud widget by describing it as a valuable tool to retrieve expressive user feedback. Half of the managers provided critical responses toward the usefulness of the EmotionKit widget, as they questioned its reliability. With only one neutral response, all managers agreed that making runtime personas available through the PersonaKit would benefit their development process.

Usefulness of FeatureKit Widgets

With respect to FeatureKit, we asked the managers on their opinions regarding widgets that visualize the feature path observations (**Q1.1**), unique devices per feature path observations (**Q1.2**), and feature crumbs observations (**Q1.3**). Overall, with one exception, we received many feedback for the free-text question **Q1.4** as depicted in Figure D.9 that asked managers to provide more insights to justify their Likert-scaled responses. Note that one manager rather elaborated on the fact that it takes a lot of time to prepare the feature crumbs—a process that precedes the visualization within any of the widgets. Two other managers that mostly agreed with all three questions and pointed out that is the combination of them that makes the individual widgets powerful.

With respect to **Q1.1**, four managers highlighted that they perceive the pie chart visualizing the feature path observations as useful to quickly grasp and understand the feature observations, i. e., it serves as a quick overview at the beginning of their analysis process. With respect to their main feature, one of the managers noted that, in case the chart does not show any indications that the user ever started the feature, they would use this information for further investigation. One other manager stated that they use this chart as a communication tool for other team members to argue for a solution. Eventually, one other manager criticized that the labels are not self-exploratory and need further explanation.

Concerning **Q1.2**, managers pointed out that the visualization of unique devices per feature path observation reflects a good extension to other widgets that deal with feature crumbs. Most of the managers brought in the example of layout issues that are related to the screen size of different devices and stated that such a widget would be very helpful in finding reasons for user's behavior. One manager provided a neutral response as the descriptions that distinguish the devices were not expressive enough; in addition, they proposed to also show the version of the operating system in the same way.

The widget that lists individual crumb observations was well-perceived and further described by four managers following **Q1.3**. They stated that they see a useful application of the widget to understand the effects and interdependency with other features. As a long-term use case for the widget, they might use its usage data in order to make decisions whether or not a feature should be deleted or changed.

Four managers provided ideas on further widgets that are based on the FeatureKit usage data; two managers wished to see how much time users spend on the interaction with a feature crumb, which would help them as **Cuu** engineers to understand how the users accomplish certain tasks. One manager reported on a widget that would be useful to them that shows the order of triggered crumbs, independent from the defined feature path. Finally, one manager imagined a heat map to visualize successfully completed feature observations.

Usefulness of ThinkingAloudKit Widgets

Turning the focus to managers' impressions on **Q2.1**, only one manager disagreed with the statement; they noted that this widget would unveil a channel for users to raise complaints which would be handed to the developers unfiltered—something they want to avoid. All other managers highly welcomed the ThinkingAloudKit widget with the majority of them agreeing with the statement. Two managers agreed with the statement without providing further insights on the usefulness. With one neutral response, three managers found the widget helpful, while they also advised caution, as the simplified visualization of the summarization of verbalized thoughts may misled developers into assumptions while the underlying classification may be wrong. In particular, they pointed out an interest in getting more insights on the grey, i. e., neutral responses, by the users. Four other managers elaborated on possible improvements for the widget; one of them requested the combination with other feedback sources within the widget, such as the enrichment using video recordings. Another manager would like to see the classification mapped to the transcribed words to get a closer look on the users' thoughts. Two managers expressed their wishes for the ability to let users summarize their feedback with a final statement at the end of a thinking aloud session.

Usefulness of EmotionKit Widgets

With only five managers that generally agreed or strongly agreed with the statement about the EmotionKit widget, this widget's usefulness perceived critically by the other half of the managers. First and foremost, one manager who strongly disagreed with the usefulness based their decision on the fact that the application relies on interactions that are triggered by the users' gaze—rendering the detection of emotions through facial expressions nearly impossible.

Two managers that disagreed and one manager that agreed with the usefulness speculated that emotions are not a suitable means to measure the user's acceptance of a feature or a usability aspect. They based their response on the fact that users (a) may be talking to others while interacting with the application, (b) may be an individual that generally shows more or less emotional responses, (c) may show a reaction that was triggered from the content rather than from the application itself, or (d) may generally show no emotions during the interaction with an application. Six managers that agreed with the statement or provided a neutral response focused their feedback on providing suggestions for further feedback. One of them acknowledged that–while they liked the idea—they were convinced that it takes some time to get used to understanding the widget, in particular with respect to the separation into four quadrants. Another manager that provided a neutral response linked the usefulness to the combination with other widgets. Three other managers suggested to provide more help in understanding the emotions, such as introducing colors for positive and negative feedback as well. Finally, one manager supposed that the widget might be confusing if there are significantly more results than three users and suggested to introduce the option to filter for certain aspects.

Usefulness of PersonaKit Widgets

Almost all managers were convinced by the usefulness of the PersonaKit widget and the visualization of runtime persona as asked by **Q4.1** One manager formulated the request of having a *click per minute* field, which—to their opinion—highly contributes to the separation of a user group. Another manager highlighted the use of the widget to identify a group of users that are happy with using the application—based on the insight from an EmotionKit widget. Two managers preferred to have the generated text presented in a more structured way, such as bullet points or subsections, rather than a block of text; generally, they expressed that the widget shows too much text. One manager noted the value of the widget when the users are from outside of the development team. Another manager pointed out the value of having instances of personas created automatically, as the traditional way of creating them is time-consuming. Two other managers welcomed the PersonaKit widget while they noted that it may conflict with data collection policies, or reflect information that is already available in form of other documents. Finally, two managers described that this information is valuable for much more activities beyond the actual development, such as for the marketing department or the user experience research.

12.2.6 Future Use of CuuSE

Following RQ 25, we investigated the managers' intention to continue using **CuuSE** after this semester. We omitted the description of answer to **Q7.3**.

Using only a free-text question for **Q7.1**, we asked the managers what other features they expect from CuuSE that are not implemented yet. One manager highlighted performance updates, as for them the use of CuuSE led to performance reduction in their application. Two managers noted to require the use of CuuSE beyond feature branches in order to make use of it in the future. Three managers describe extensions to the way feature crumbs can be used, such as a visual path representation of crumbs, time tracking of crumbs, or the nesting of crumbs. One manager noted that it should be easier to setup the Cuu platform, while another one asked for compatibility for multiple devices that perform one feature path. Finally, one manager requested gaze support for analyzing where the user is looking on the screen.

As part of **Q7.2**, we asked managers of additional widgets that would be useful to them. Five managers did not provide specific suggestions. Two managers describe an improvement for the feature crumbs widget that would require developers to enrich the crumbs with more information; for example the input from sensors, such as from an external device, or descriptive messages, such as *"Wrong user input"*. The visualization of this information within a dedicated widget would broaden the scope of use cases that could be addressed with CuuSE. Similar suggestions pointed toward allowing multiple entry points for feature paths. One manager continued their thought from the previous question and described how face tracking would be beneficial to track the usability of their augmented reality application. Finally, another manager reported that a widget that shows app crash reports would be helpful to them. The remaining questions were supported by Likert-typed questions. Table 12.6 depicts the managers' responses.

Table 12.6: Managers' intention to use the CuuSE framework.

ID	Question (see Figure D.13)	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Q7.4	I would add feature crumbs to my applications' source code to enable feature path observation visualization as shown in the widgets from Q1.	1	7	0	1	1
Q7.5	I would add, modify, or extend existing classes in my applications' source code to enable additional widgets in CuuSE, such as the Emotion widget shown in Q3.	0	4	3	1	2
Q7.6	CuuSE supports teams during usability engineering.	2	5	1	1	1
Q7.7	I would use CuuSE in future projects.	1	8	0	0	1

With respect to **Q7.4**, from the ten managers, eight agreed that they would add feature crumbs in their future projects in order to benefit from the visualizations. However, from **Q7.5** it becomes obvious that only four of the managers would be willing to add further modifications to their application in order to benefit from knowledge sources such as EmotionKit[40]; with three managers that provided a neutral or disagreeing statement, respectively. These numbers can be seen in a different light after analyzing the responses: Four managers unconditionally agreed to the statement. Two managers noted a technical bug with the EmotionKit, which prevented them from applying the changes; they stated that—if these issues would be resolved—they would start using it in the future. Two other managers explained their non-agreeing response with the fact that they question the validity of EmotionKit; if they would be assured that it works as described, they would be adding the modifications right away. One manager who strongly disagreed with the statement justified their response on the basis that they do not know what the source code changes are good for. One other manager provided a neutral statement as their application requires two devices interacting to complete a feature.

Most of the managers agreed that CuuSE supports teams during usability engineering (**Q7.6**). One neutral response was not justified. The two disagreeing statements were motivated by answers to previous questions, as one manager stated that CuuSE does not fit their needs. The other manager noted that, while they think it is helpful for small teams, larger teams would find it confusing. The remaining managers agreed or strongly agreed with the statement while quoting previous request for improvement, such as the extension to other branches despite feature branches or iterative feature paths. One of them reiterated on some of the bugs which reduced the ease of use of CuuSE.

In the last question **Q7.7**, with one exception, all managers agreed or strongly agreed that they would use CuuSE in future projects. The one strongly disagreeing statement was based on the fact that the team did not have any users besides themselves and therefore did not use CuuSE intensively; which put the manager into a position from which they could not make any prediction whether they would use it in the future. One manager who agreed with the statement highlighted that CuuSE would provide an objective way to judge users' behavior; and would therefore continue using it. Four managers generally agreed and strongly agreed with the statement by listing the benefits for them; four other managers noted that they would make the decision dependent on the size of the project and the future users.

[40]This relates to using customized classes for standard components, such as views, in order to continuously capture camera inout. We provided more insights into the rationale for code modifications as well as the implications for the managers in Section 8.1.2.

Summary RQ 25: Overall, almost all of the managers stated to intend to use **Cuu^{SE}** in future projects. They were willing to add feature crumbs to the source code to benefit from the widgets; provided that some technical issues are resolved and the validity of the usage knowledge is guaranteed, they also intend to add, modify, and extend existing classes in their application. The managers provided suggestions for improvement to ensure their future use of **Cuu^{SE}**. On the one hand, to increase the usefulness, the managers listed multiple extensions, such as knowledge sources for gaze and crash reports, as well as control of feature releases. On the other hand, with respect to the ease of use, their mainly requested improvements in form of stability and performance improvements.

12.3 Discussion of Results

In this section, we provide a brief interpretation of the results (Section 12.3.1) and outline threats to the validity of the survey (Section 12.3.2).

12.3.1 Interpretation

From the managers' responses, we derived the following five topics of interest which significantly affected their perceived usefulness, ease of use, and intention to use **Cuu^{SE}**. We summarize the responses and provide an interpretation.[41]

Acceptance of Cuu Platform

From the responses across all questions, we can derive that the managers show a high acceptance of the **Cuu^{SE}** and the **Cuu** platform. Following **Q1/Q1.1**, most managers agree or strongly agree that such a platform is useful to establish a common understanding of feature within a development team. We understand this as a major endorsement for the platform and the continuous user understanding platform.

From **Q1/Q2.1**, **Q1/Q2.2**, and **Q1/Q2.3**, we also derive that the setup of the platform does not cause major efforts for the managers—one aspect that we expected to be more difficult. This may be understood as a confirmation that the **Cuu** syllabus is a suitable means to help managers to setup the platform. It also highlights the close relationship between the validation presented in this chapter, as well as the validation of the **Cuu** syllabus, allowing the conclusion that results from one can be applied to the other and vice versa.

The fact that the majority of the managers responded with agreeing statements, when being asked for the usefulness of **Cuu^{SE}** during usability engineering (**Q2/Q7.6**)

[41] Note that due to the same numbering within both questionnaires, we add the prefix Q1 or Q2 to the number of question we are referring to in order to distinguish between questions of different questionnaires, e. g., **Q1/Q1.1** or **Q2/Q7.3**.

as well as whether they would use **Cuu**$^{\text{SE}}$ in future projects (**Q2/Q7.7**), provides further evidence that the platform supports developers in establishing a continuous user understanding activity, answering Knowledge Question 5.

Technical Issues with Cuu Platform

Across multiple questions, the easy of use of the platform was criticized by managers due to technical issues, i. e., slow responses and downtimes: This was in particular notable during the creation and management of features (**Q1/Q3.1** and **Q1/Q3.3**) or when loading new usage knowledge (**Q1/Q4.1**).

Also, we observed from responses to **Q2/Q7.5** that multiple manages were not able to use all functionality offered by the **Cuu** SDK due to the fact that some required code modifications were incompatible with functionality they were using, causing an internal bug; this affected applications that were using ARKit, the technology that was also used to monitor facial expressions for EmotionKit, as described in Section 8.2.1. In addition, **Cuu**$^{\text{SE}}$ is not compatible with features that require the interaction between two or more devices (**Q2/Q7.5**).

We acknowledge that the use of a beta version of **Cuu** platform caused multiple issues due to its level of maturity. This is also reflected in multiple suggestions for new widgets requested by the managers (**Q2/Q7.1**). While we expect that most of the above-described technical issues can be resolved, we understand some of them as limitations that relate to the overall concept, such as the support of more than one device for feature observations.

Complexity of Feature Crumbs

The managers generally acknowledge that linear feature paths are useful to analyze the usability of a related feature (**Q1/Q1.3**). In fact, they expected the platform to offer such as mechanism in order to track usage knowledge (**Q1/Q4.2**) and are willing to add it to their source code (**Q2/Q7.4**). However, they verbalized multiple extensions to the feature crumbs concept as many of their features cannot or only partially be reflected using the linear feature paths (**Q1/Q4.2**), such as adding loops or decision crumbs, or other additions (**Q2/Q7.2**), such as adding meta information to the crumbs. At the same time, it is already difficult to find the right spot in the code to place the feature crumbs, as two managers reported **Q1/Q3.2**. One manager found a good explanation in calling it a trade-off that must be found in the complexity of the feature crumbs, e. g., a detailed and realistic recreation of the actual feature, and the usability, i. e., simple integration into the source code and assessment of the usage knowledge results. This aspect is also expressed in the managers' answers that addressed the use of feature branches: they asked to use the concepts also on develop and main branches **Q1/Q1.2**.

We made a design decision when limiting feature crumbs to a linear sequence, as this facilitates the visualization on usage knowledge in the widgets. We understand that this excludes some features from being able to be observed using CuuSE; we argue that these features need to be reduced to even more fine-grained parts that can be depicted using a linear sequence. At the same time, we consider adding new crumb types that are generally compatible with a linear flow; such as the implementation of optional crumbs as described in the concept in Figure 7.4, or the availability of loops, i. e., allowing a crumb to be triggered multiple times without causing the feature path to be cancelled.

Expressiveness of CuuSE Artifacts

Based on **Q2/Q1.1** and during multiple other questions that address widgets, we understand that the visualizations require more support in their understanding. While we plan to add more descriptions in the widget's canvas, we already added an information window that can be opened using the icon placed in the top right corner, as it can be seen from Figure 7.15. The modal windows show more information on the widget's content and how it can be used to gain more insights. Similarly, the managers asked for minor visual improvements throughout the platform, such as adding information on the color of commits or providing a more intuitive relationship between springboard and widget, as well as focusing more on the content of the widgets than on the springboard (**Q1/Q4.3**). We plan to iterate on these aspects and add improvements if required.

Understanding of CuuSE Framework

With no disagreeing statement, the managers appreciated all three FeatureKit widgets and highlighted its applicability to derive an initial understanding of the users' response toward the feature (**Q2/Q1.1**, **Q2/Q1.2**, and **Q2/Q1.3**). Other widgets received multiple feature requests, such as for the ThinkingAloud widget (**Q2/Q2.1**) and for the PersonaKit widget **Q2/Q4.1**, while the usefulness of the latter one was particularly welcomed. Multiple concerns were raised regarding the validity of the usage knowledge presented in the EmotionKit widget (**Q2/Q3.1**). We suppose that this was due to the fact that most of the managers were not actively using this functionality and could only base their responses on the descriptions provided in the questionnaire. Generally, we observed that many managers were requesting functionality that is already available in CuuSE. However, it requires more knowledge about the platform itself, such as when depicting the temporal list of triggered feature crumbs as requested in (**Q2/Q7.1**), which is implemented in the usage data trace widget as presented in Figure 8.3, or other combinations of widgets that could be created by adding multiple instance to the dashboard using the add button as de-

picted in Figure 7.15-Ⓜ. We argue that this understanding is increased over time after having applied **CuuSE** in practices.

12.3.2 Threats to Validity

This section briefly outlines the threats to the validity of the presented survey centered around the four dimensions of validity [265], namely construct, internal, and external validity as well as the reliability.

Construct Validity

Overall, we posed 60 questions to the managers spread over two questionnaires, while each one might be understood differently by the managers than intended by the researchers. While this risk is lower for the 27 free-text questions, as they allow assumptions on the basis of the provided text, the risk is higher for the 23 Likert-typed questions. We tried to mediate this threat to the construct validity by allowing the managers to ask questions at any time; supported by **CuuSE** coaches who distributed the questionnaires to the managers, asking for help was possible, and reducing the chances of misunderstandings.

Internal Validity

During the performance of the survey, there might be the risk that other, external, or third factors included the perception of the managers' that influenced their opinion toward **CuuSE**, even though it has not been related to it. We observed multiple technical issues that were related to the applications' code that—while they were not related to or caused by **CuuSE**—affected the mood of the managers toward it. We tried to mediate this threat to internal validity by providing visionary scenarios during the questionnaires, to which the managers could relate their answers to, in order to let them influence their answers by factors related to their individual application.

External Validity

The generalizability of results is an important aspect which should be addressed by the survey. Since we only conducted the survey during one semester, it may be possible that we would retrieve different results in another instance of the iPraktikum. We argue that the composition of projects represented a typical instance of the iPraktikum, which supports the validity of the results. Further, we highlighted answers that we perceived as outliners, i. e., once that significantly deviated from what one could expected as an answer—as observed with one manager and noted when reporting the first results in Section 12.2.1.

Reliability

The results of the survey were analyzed by only one individual researcher. This poses a threat to the reliability of the results. We tried to mediate this threat on two ways. First, almost half of the questions contained Likert-typed questions, which do not leave much room for interpretation. Second, with respect to the free-text questions, we clearly described the chain of reasoning when describing the results of the managers' responses; this should allow other researcher to come to the same conclusions as we did.

Part V
Epilog

THE EPILOG completes this dissertation. We provide a conclusion of the presented work, which comprises of a summary of all knowledge questions from the problem investigation and treatment validation, as well as the artifacts that resulted from the treatment design. The conclusion further addresses privacy considerations in the context of **Cuu^{SE}**, our proposal toward automatic the extraction of tacit knowledge, as well as visionary ideas for adapting **Cuu^{SE}** to domains other than software evolution. We close the dissertation with license information and supplemental material, as well as a list of both figures and tables, and the bibliography.

Chapter 13

Conclusion

> *"Will passengers be able to have meaningful conversation with their cars? In the past, the human tendency to assign beliefs, emotions, and personality traits to all sorts of things has been criticized as anthropomorphism. As machines gain in their cognitive and emotional capacities, the anthropomorphism may not be so erroneous. These assignments might very well be appropriate and correct"*
>
> — DONALD A. NORMAN [225]

This chapter concludes the dissertation. Section 13.1 summarizes the main contributions of our work. Section 13.2 revisits the aspects of privacy for continuous user understanding, how it is addressed by **Cuu^{SE}**, and further ways to approach it. Section 13.3 describes the possibility of automatically extracting tacit knowledge. Section 13.4 discusses how **Cuu^{SE}** can be adopted by other domains and how it could thereby contribute to user understanding in these domains.

13.1 Summary

We summarize the contributions of the dissertation with respect to the Knowledge Goals, the Technical Design Goal, and the Instrument Design Goal, as introduced in Section 1.4, which served as the basis for the design science approach.

Problem Investigation

We presented an introduction to Continuous Software Engineering (CSE) and introduced its classification into CSE elements and CSE categories. We outlined a vision for integrating knowledge into CSE, which originated from CURES research project. Hereafter, we reported on the design of a comprehensive case study that addressed three knowledge questions in the context of this dissertation: Using semi-structured interviews, we gained insights from 24 practitioners from 17 companies who shared

their experiences from 20 industry projects. On the basis of the following results, we addressed the Knowledge Goal 1.

Current Practices in CSE

Practitioners exhibited different definitions of CSE; we identified a tool, methodology, developer, life cycle, and production management perspective when elaborating on their definition of CSE. Practitioners perceived CSE elements from the quality, user, and developer category as most relevant, with a focus on CSE elements that enable the basic operation of CSE, such as version control management. We recorded 19 positive, 56 neutral, and 17 negative experiences in the practitioners' answers. The code and software management categories received many positive mentions, which qualifies them as starting points for establishing CSE. Most of the negative experiences addresses the developer category. Practitioners' future plans for CSE are vague; only 19 CSE were addressed by them. We identified three main strategies which we summarized as enhancement, expansion, and on-demand adaption. Eventually, we derived the *Eye of CSE*, a model to describe CSE as well as to provide support in both establishing and advancing CSE in a company.

User Feedback in CSE

With respect to the involvement of user in CSE, the practitioners rely on explicit user feedback. While none of them relies solely on implicit user feedback, almost half of them use it to support explicit user feedback. There is no dominant strategy for practitioners to which artifact they relate user feedback to, however, they struggle to establish usage knowledge by relating the user feedback to a feature under development. Most of the practitioners do not continuously capture user feedback. While more than half of the practitioners employ tool support, a similar number also makes use of manual capture processes, which reduces their efficiency. Feedback from external sources dominated, however, internal sources promised to revel valuable insights. Technical aspects hinder practitioners to capture the context of user feedback. Practitioners rely on the user feedback to verify requirements, rather than its validation. Only a limited number of practitioners exploits changes in the user feedback over time and combines different user feedback types.

Usage Knowledge in CSE

The practitioners' attitude toward our vision of integrating usage knowledge into CSE is predominantly positive: with respect to their overall impression, we recorded 1 negative, 3 neutral, and 15 positive answers. Regarding the overall feasibility, 11 neutral impressions dominated the answers following 7 positive and 2 negative answers. In the practitioners' answers, we identified five benefits regarding

the visions' proposals for usage knowledge, i. e., accountability, traceability, parallel feedback, automation, and flexibility. Obstacles addressed user groups, applicability, usability, and privacy. Following their answers, the vision can be extended in the context of automation, roles, run-time aspects, as well as human and processes. The practitioners also proposed additions to the vision that address its integration with other components, experimentation, interaction and reaction aspects, the granularity of knowledge, as well as to the focus on developers. We summarized the practitioners answers in an improved version of the CURES vision.

Treatment Design

The **Cuu**^{SE} framework supports developers in understanding users' needs during CSE. We exemplify how its individual building blocks contribute to this goal using the metaphor of debugging source code:

> **Example 11:** The **Cuu** workbench acts as the equivalent of an Integrated Development Environment (IDE). It supports engineers in managing feature crumbs, which—similar to break points—are manually added to a line of code that is of interest, i. e., where they suspect a particular of a feature that requires further investigation. IDEs provide different tools to inspect the state of a program after it has been stopped by a break point; for instance, visual representations of a method's call stack or the content and type of variables. This supports developers in program comprehension. Similarly, **Cuu** kits act as domain-specific tools that allow engineers to gain valuable insights, however, with respect to the users' behavior and thoughts. They also present this usage knowledge visually.

The **Cuu**^{SE} framework provides tools to effectively capture and extract user feedback. We argue that the process of examining a feature under development is key to understand the users' needs and establish a comprehensive understanding of users. On the basis of the following artifacts, we achieved the Technical Research Goal as well as the Instrument Design Goal.

Feature Crumbs

We introduced feature crumbs as a lightweight, code-based concept to map usage data to the particulars of a feature. This enables the creation of a usage traceability, a main requirement for adapting usage monitoring to CSE processes. Feature crumbs are reflected as single-line additions to the application's source code. Two or more feature crumbs form a feature path that contribute to the gathering of knowledge about a feature's observation state, i. e., whether it has been started, canceled, or finished by the user.

Cuu Workbench

We presented the **Cuu** workbench that comprises of both a dashboard and a platform. The dashboard enables the visualization of usage knowledge using two main abstractions: springboards, which serve as knowledge selectors, and widgets, which visualize the respective knowledge. Widgets further distinguish into three classes—spot, compare, and range—which enables multiple different knowledge inspection scenarios. The dashboard allows the management of feature crumbs and in particular the definition of feature paths. The platform component ensures the integration of the workbench into existing CSE tool chains. This provides the ability to connect the workbench to version control systems to automatically receive updates on new feature increments, as well as basic handling of **Cuu** projects and user management.

Cuu Kits

We demonstrated the extensibility of **Cuu**$^{\textbf{SE}}$ with the implementation of seven knowledge sources, so-called **Cuu** kits. Each of them encompass the competence to collect or monitor usage data of varying types. Furthermore, **Cuu** kits feature the expertise to process it to usage knowledge. FeatureKit, InteractionKit, and NoteKit represent low-level knowledge sources that focus on the collection of usage knowledge that is of use for other kits; i. e., the observation of feature paths, the monitoring of user and application interactions, as well as the ability to collect explicit user feedback. EmotionKit is a knowledge source that provides access to users' emotions on the basis of observed facial expressions. ThinkingAloudKit automates the Thinking Aloud protocol by recording and classifying users' verbalized thoughts whenever they start using a new feature increment. BehaviorKit enables the classification of multiple attributes of individual users on the basis of how they interact with the application. PersonaKit introduces the concept of a runtime persona, which allows the classification of user groups on users' interaction and application meta data.

Cuu Workflow

We developed the **Cuu** workflow as a description of how the individual artifacts of the **Cuu**$^{\textbf{SE}}$ framework can be applied by developers to extract tacit knowledge. The core activity of the workflow lies in the analysis of usage knowledge: following a funnel approach, we propose to incrementally add and remove usage knowledge widgets to the dashboard in order to understand the users' needs and work toward a solution for an improvement.

Cuu Syllabus

We designed the **Cuu** syllabus in order to disseminate the artifacts of the **Cuu**$^{\textbf{SE}}$ framework. We focused on the introduction within an academic context. We introduced

the roles of a **Cuu**^{SE} manager and a **Cuu**^{SE} coach. Following a series of four meetings, the syllabus enables individuals to employ the **Cuu**^{SE} artifacts.

Treatment Validation

We applied three empirical cycles to validate the artifacts of the **Cuu**^{SE} framework. On the basis of the following results, we addressed the Knowledge Goal 2.

EmotionKit Validation

In a laboratory experiment with 12 participants, we validated the applicability and feasibility to detect usability problems in a sample application. In a qualitative analysis of the results, we found that the measurements reflect our manual observations and individual peaks can lead to the occurrence of usability problems. To address the issue of noise in the data, we performed a quantitative analysis of the measurements: we calculated the emotional response of participants which serve a better mean to detect usability problems. In the manifestation of their amplitudes, we identified phases that described the start, peak, and end of emotional response, which, on average, tool approximately 2, 3, and 7 seconds, respectively.

Syllabus Validation

In a quasi-experiment with 9 student developers, we validated the applicability of the **Cuu** syllabus to disseminate the **Cuu**^{SE} framework. Following the Goal Question Metric approach, we identified three questions and nine metrics for the validation. We concluded that the syllabus enables the introduction of the main concepts in the scope of one semester and supports the developers in creating feature representations. While the syllabus facilitates the use of **Cuu**^{SE} artifacts, we found potential for improvement with respect to its long-term usage.

Framework Validation

In a survey that comprised of two questionnaires with a total of 60 questions, we validated the combination of multiple **Cuu**^{SE} artifacts with 10 student developers. Following the Technical Acceptance Model, we derived their perceived usefulness, perceived ease of use, and intention to use **Cuu**^{SE} with respect to multiple aspects: the concepts for user understanding, the setup process, the management of features, the analysis of features, individual **Cuu** widgets, as well as the future use of **Cuu**^{SE}. For the subset of quantitative survey questions, we applied the Likert scale; we received the following distribution of answers: 52 strongly agree, 103 agree, 38 neutral, 18 disagree, and 8 strongly disagreed.

13.2 Privacy Consideration

We introduced **Cuu**^{SE} as a framework for continuous user understanding. This makes the collection of information about the user inevitable, which raises questions regarding the users' privacy. **Cuu**^{SE} strives to enable an understanding—rather than the tracking—of users. Therefore, we limited the application of the framework to a selected group of users who are aware that they are being observed and that the data they provide is used for user understanding.

The answers of practitioners in the interview study suggested that company-internal sources can act as a helpful provider of user feedback as well. Therefore, fellow developers—or the developers of a features themselves—might qualify for an application of **Cuu**^{SE} as well. Their work with **Cuu**^{SE} on a daily basis can contribute to a deeper understanding of the underlying technology which may results in fewer concerns about the privacy and trust in the application of the framework.

However, as end users may be involved in the data collection and utilization using **Cuu**^{SE}, we provided multiple means to ensure misuse of **Cuu**^{SE}. On the one hand, as described in Section 8.1.2, we inform the user about the integration of the **Cuu** SDK as soon as they start the application. In case they want to refrain from providing a particular type of feedback, they can deselect this aspect as visualized in Figure C.1. On the other hand, we developed concepts such as the runtime persona, that acts as a cluster proxy, which obfuscates particulars of an individual user.

Note that the design of the kits plays a crucial role in our endeavor to achieve continuous user understanding without compromising the users' privacy. For example, the collection of facial expressions and extraction of emotions is a highly sensitive process with consequences for the users' privacy. This becomes even more relevant as consumer hardware promises to improve sensor accuracy. User tracking techniques on the basis of facial features demonstrate the potential for misuse of the such data. This is why platform providers should restrict the usage of such techniques, which—in turn—restricts the applicability of EmotionKit in practice.

In general, as it becomes clear from the validation results in Section 10.1, the capture of emotions provides great value for software engineering. We do not want to fully disclose the users' data, yet we still want to leverage the beneficial technology. This raises the need to establish a policy when working with user data. Therefore, we propose a rule, the *functional aggregation*, to process data:

> **Functional Aggregation:** User-specific information shall only be transported and utilized in an aggregated format given an application-specific purpose.

When we apply this rule to the EmotionKit, we assume that the transformation from face features to action units and then to emotions should be performed on the device itself at the point of its origin. Only information about the emotion is trans-

ferred and made available, as it was determined as the application-specific usage of this information elicitation. The emotions form a container format for a specific purpose and obfuscate the underlying data. The need for an application-specific purpose poses the requirement to developers to define for which usage knowledge goal the data is collected. It blurs the fact of how an information was collected, adds uncertainty about the source, and ensures that future misuse of the original core data set, i. e., individual measurements that lead to an assumption, is avoided. The latter aspect is relevant that, given the availability of a new technology or additional context information, it would be possible to draw a conclusion for other application-specific purposes. The functional aggregation of user data benefits from emerging technology, such as fog computing [40].

13.3 Toward Automatic Tacit Knowledge Extraction

In its current version, **Cuu**^{SE} follows a semi-automatic procedure: While the management of usage data in **Cuu**^{SE} is performed automatically, we still require the developer to inspect the results of usage knowledge manually in a dashboard. To support this process, we provided the **Cuu** workflow that guides the developers toward the extraction of users' tacit knowledge.

In future versions of **Cuu**^{SE}, we imagine to reduce the influence of a `Controller` class in Figure 7.1 that currently facilitates between the **Cuu** workbench and the individual **Cuu** kits. The blackboard architecture allows to promote **Cuu** kits to knowledge sources that independently, autonomously, and collaboratively work toward the goal of extracting tacit knowledge. While this collaboration may start on hierarchical basis in which each kit on its own develops low-level usage knowledge in isolation, they may start communication to gain the ability of incorporating other usage knowledge to improve their own results, as described in Figure 7.2. The communication can be enabled by a new knowledge event model that allows kits to subscribe to new usage events. For instance, new insights created by EmotionKit can contribute to a better distinction of runtime personas by PersonaKit.

To develop **Cuu**^{SE} toward a system that allows automatic tacit knowledge extraction, we need to introduce a system that follows a non-linear process structure. In doing so, traditional personas [64] serve as the starting point. They capture a users' characteristics and encapsulate the observations that result of an asymptotic process of knowledge extracting. This could be a *greedy* algorithm which starts with a local optimum as the initial state of a hypothetical stakeholder. For the purpose of summarizing and allocating usage knowledge, we rely on the concept of runtime personas presented in Section 8.5.1. According to Polanyi, tacit knowledge is a person-related concept [248], which is reflected in the definition of personas. As the **Cuu** kits continue to communicate and enrich the runtime persona with their

usage knowledge, the system is moving toward a better representation of the users' tacit knowledge. Adapting Polanyi's theory of the phenomenal structure of tacit knowledge, users are aware of their interactions from which they are attending to accomplish the feature—in appearance of that specific feature. Therefore, we propose to utilize a modal window that asks the user for explicit feedback. It is triggered as soon as the **Cuu** kits agree that the user is about to apply tacit knowledge. We suggest to ask the user what they were about to do, how they tried to achieve it, and whether they experienced any issues so far. The results are presented to the developer—in combination with the measurements of the **Cuu** kits—in a dedicated view of the dashboard. This enables them to understand the tacit knowledge.

The automatic extraction of tacit usage knowledge faces various challenges. For once, the kits need to be able to distinguish error conditions and noise from actual user behavior that is relevant for the tacit knowledge. This leads to the question whether every user behavior is relevant, and whether or not there is such thing as noise? In addition, with respect to refer to the user as a sources of explicit user feedback to provide the rationale for their actions, a challenge lies in the question when a user can be interrupted. We argue that two requirements need to be met: First, a user should only be interrupted if it can be guaranteed that it will not interfere with their current workflow. Second, no critical process should be disturbed. Both requirements, however, pose new challenges. A balance needs to be found to keep a minimal time span between the interaction and the interruption. A delay, though, results in the problem that traceability should be guaranteed, i. e., the users' feedback should be mapped clearly to an interaction. To address this issue, we envision to develop a tacit knowledge characteristic similar to the properties defined in database transactions: atomicity, consistency, isolation, and durability [134].

13.4 Adapting CuuSE to other Domains

So far we presented **Cuu**$^{\textbf{SE}}$ in the context of software evolution with a focus on mobile applications. In the following, we sketch visionary ideas of how the framework could be adapted to other domains.

Driving Environments

Modern cars are equipped with a series of sensors that allow usage data to be collected and processed from the way the driver interacts with a car's interior, such as physical knobs, voice assistants, or graphical user interfaces such as the car's multimedia unit. In addition, researchers try to understand the drivers' behavior by combining vehicle, driver, and environmental data with the goal to understand scenarios of accidents [242].

Similar to our descriptions in Chapter 8, we imagine to create different kits that operate within a car for usage data capture and processing of usage knowledge; this may be the adaption of existing **Cuu** kits, such as:

- EmotionKit to derive emotions from facial expressions. Fridman *et al.* demonstrated the use of cameras to monitore the driver and derive situations of emotional stress [113]. While driver-facing cameras are step-wise becoming available in cars, the addition of three-dimensional cameras might further improve their effectivity.
- InteractionKit to record any physical interaction, which serves as a trace for a detailed analysis of actions.
- ThinkingAloudKit to provide the ability to drivers to verbalize explicit feedback. The fact that ThinkingAloud kit relies on verbal feedback promotes the use in the car, as it is less likely to disturb the driver from steering the car. A possible extension might be the use of a dedicated button which allows drivers to start recording the feedback.

Additional kits could also be developed that are specific to cars and base on the availability of hardware sensors:

- A kit to detect the level of carbon monoxide that can be measured in a closed system to provide more insights about the driver.
- A kit to utilize the pressure sensors in the steering wheel and sensors to detect movements on the seat that reveal the drivers' physical state.

We imagine three scenarios in which a continuous user understanding, enabled through **Cuu**$^{\textbf{SE}}$ and its kits, benefits driving environments.

First, for an improvement of the in-car experience during design-time. This is similar to the use of **Cuu**$^{\textbf{SE}}$ for software evolution: Engineers inspect the collected usage knowledge in a dashboard that is part of the workbench to improve an aspect of the car for the next product cycle or feature improvement.

Second, with respect to the utilization of usage knowledge during run-time of a system, a new controller component could replace the workbench for in-situ usage of the collected knowledge. Based on the automatic detection of tacit knowledge from multiple knowledge sources, the car could automatically react to the driver's needs, such as adapting the air conditioner or sound volume.

Third, kits could also contribute to the self-driving capabilities of autonomous cars. In this case, the workbench is replaced by a training component that converts the extracted tacit knowledge into labels and combines them with input features, such as video footage, for *machine learning*. This enables a continuous training of classification models on the basis of human traits, which ultimately results in natural, autonomous driving behavior.

Living and Working Environments

With respect to living environments, InteractionKit, BehaviorKit, or PersonaKit could measure the interaction with fixtures, such as lights, blinds, doors, or windows. Sensors that provide measurements, such as motion or temperature, could be integrated into domain-specific kits and process the data similar to EmotionKit. The extraction of users' tacit knowledge in living environments could be utilized to improve the overall user experience and to increase precision in controlling specific fixtures.

With respect to working environments, an increasing number of mobile sensors are worn by individuals, such as smart bands, watches, or glasses. This enables the creation of new kits. For example, Schaule *et al.* demonstrated that smart watches provide data that serves as a suitable means to predict the cognitive load of a knowledge worker [271, 272]. Using this knowledge, **CuuSE** could act as a critical component for user understanding with respect to cognitive load.

Appendices

Appendix A

Licences

As addressed in Chapter 1, this dissertation is based on multiple publications that have been published through different publishers. For all used material, the permission for reuse in this dissertation was requested from publishers via email. Each request contained the following content:

1. The work in question.
2. A description of request, i. e., reuse in dissertation monograph.
3. Additional notes, such as that the work was published as the lead author, the intention to reuse and extend the content rather than include the full published work, the requirement of publishing the dissertation in a non-commercial form in the library of the Technical University of Munich, and notice that full acknowledgement to the published work in the respective online libraries will be provided.
4. A request whether a dedicated license process is required.

All publishers granted permission for reuse in one way or the other; the individual responses in Appendices A.1, A.2, A.3, and A.4. their responses, which describe the extent of granted permission, is summarized in the following. For all personalized responses, names and dates have been omitted, however, the core content was left unchanged.

While the author of this dissertation holds the copyright for the work [153] and [152] (more details in the first part of Appendix A.5), we also reached out for the publishers and were encouraged to reuse respective work in this dissertation. In the same way, use of work [160] did not require a permission grant as detailed in the second part of Appendix A.5.

Usage of content from Bachelor's and Master's theses was permitted by the supervised students. More information can be requested from the dissertation author.

A.1 IEEE

Content of IEEE-published work ([151, 154, 156, 159]) is used in this dissertation based on the following permission provided via email:

Dear Jan Ole Johanßen,

The IEEE does not require individuals working on a dissertation/thesis to obtain a formal reuse license however, you must follow the requirements listed below:

Textual Material

Using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © [Year of publication] IEEE.

In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.

If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Full-Text Article

If you are using the entire IEEE copyright owned article, the following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]

Only the **accepted** version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line. You may not use the final **published** version

In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to

http://www.ieee.org/publications_standards/publications/rights/rights_-link.html

to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

We retrieved the permission grants from the IEEE Xplore Digital Library as shown in Figures A.1, A.2, A.3, and A.4.

Figure A.1: Permission grant for [154].

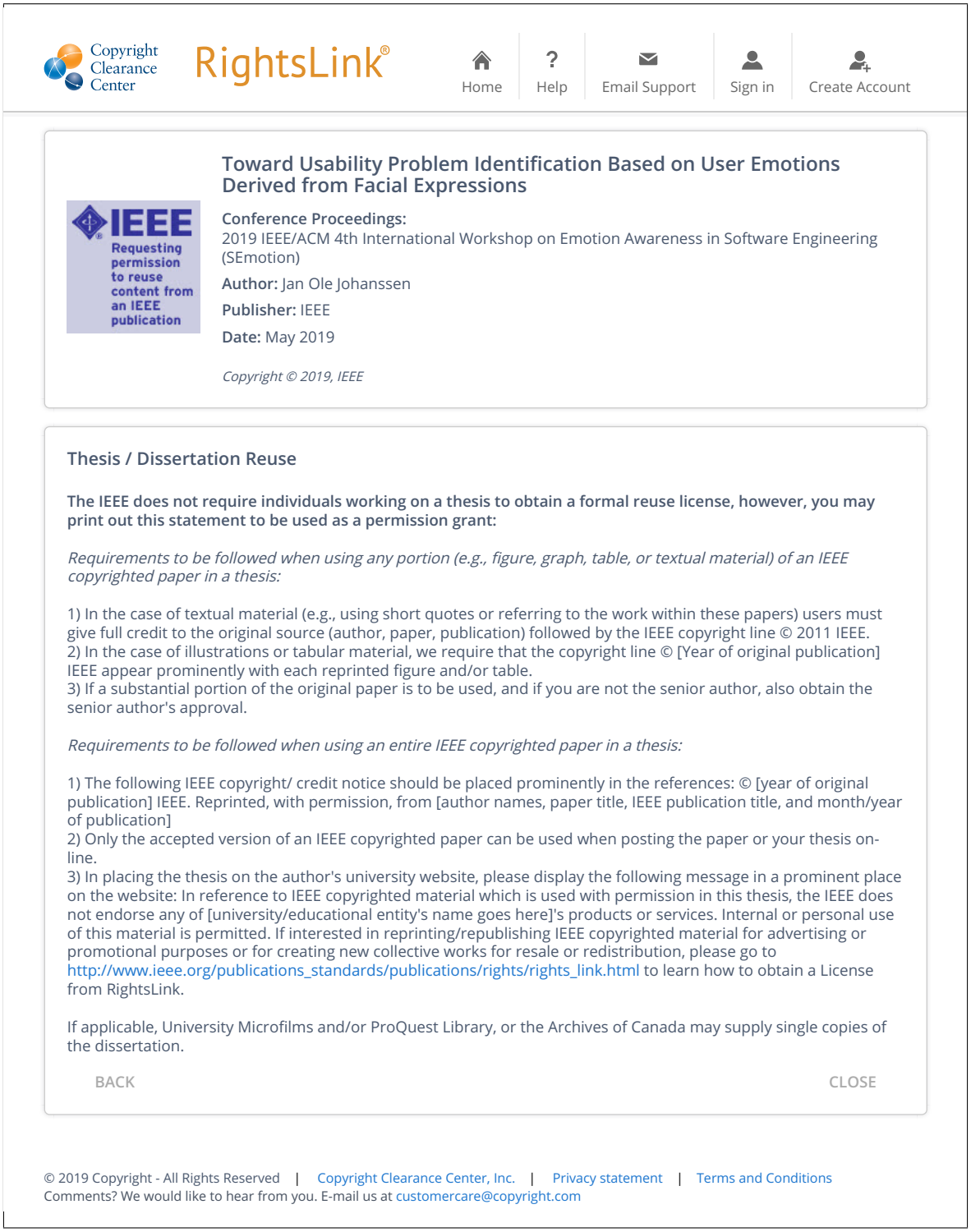


Figure A.2: Permission grant for [151].

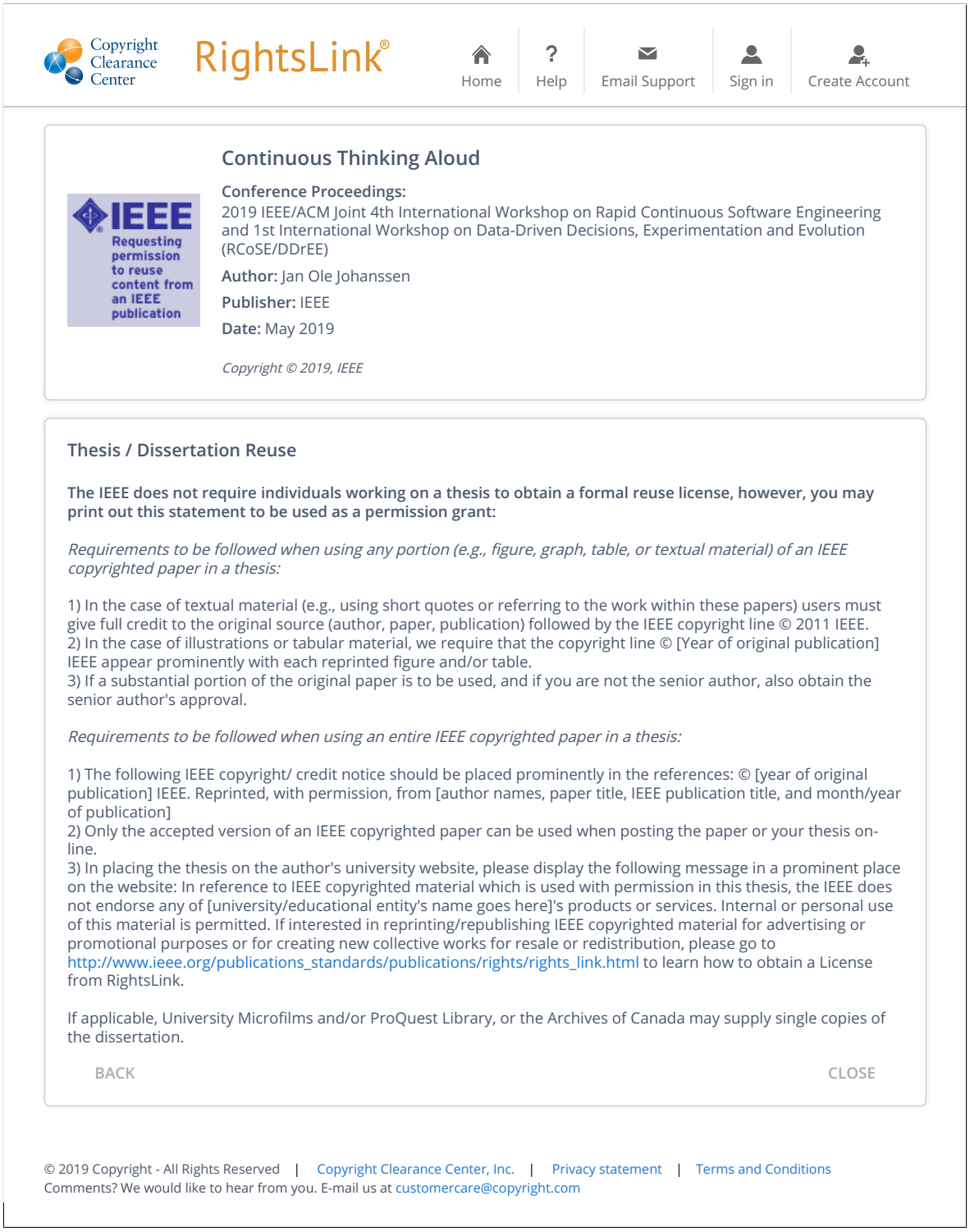


Figure A.3: Permission grant for [159].

Figure A.4: Permission grant for [156].

A.2 Wiley

Content of the WILEY-published work ([158]) is used in this dissertation based on the following permission that has been provided via email:

Dear Jan,

It is always good pratice to request permissions for any kind of reuse, we do have options avaliable via RightLink to facilitate our authors. [sic]

On this occasion permission is granted via email for you to use the material requested for your thesis/dissertation subject to the usual acknowledgements (author, title of material, title of book/journal, ourselves as publisher) and on the understanding that you will reapply for permission if you wish to distribute or publish your thesis/dissertation commercially.

You should also duplicate the copyright notice that appears in the Wiley publication in your use of the Material. Permission is granted solely for use in conjunction with the thesis, and the material may not be posted online separately.

Any third-party material is expressly excluded from this permission. If any material appears within the article with credit to another source, authorisation from that source must be obtained.

For future reference I have included the below links, these detail what you can and cannot share freely as an author;

http://olabout.wiley.com/WileyCDA/Section/id-826716.html

https://authorservices.wiley.com/asset/Article-Sharing-Guidelines.pdf

Should you require any further information, please do not hesitate to contact me.

A.3 ACM

Content of the ACM-published work ([157]) is used in this dissertation based on the ACM Author Rights which are available online[42]—an excerpt of the relevant passages is included in the following:

REUSE
Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is not the editor, requires permission and usually a republication fee.

Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).

- Commercially produced course-packs that are sold to students require permission and possibly a fee.

For the ACM-published work [150], the dissertation author holds the copyright.

[42] https://authors.acm.org/main.html

A.4 Springer

Content of the Springer-published work ([155]) is used in this dissertation based on the author permission retrieved via RightsLink (Copyright Clearance Center, Inc). An excerpt of the relevant passages of the terms and conditions is included in the following:

Springer Nature Terms and Conditions for RightsLink Permissions

Springer Nature Customer Service Centre GmbH (the Licensor) hereby grants you a non-exclusive, world-wide licence to reproduce the material and for the purpose and requirements specified in the attached copy of your order form, and for no other use, subject to the conditions below:

1. The Licensor warrants that it has, to the best of its knowledge, the rights to license reuse of this material. However, you should ensure that the material you are requesting is original to the Licensor and does not carry the copyright of another entity (as credited in the published version).
2. If the credit line on any part of the material you have requested indicates that it was reprinted or adapted with permission from another source, then you should also seek permission from that source to reuse the material.
3. Where print only permission has been granted for a fee, separate permission must be obtained for any additional electronic re-use.
4. Permission granted free of charge for material in print is also usually granted for any electronic version of that work, provided that the material is incidental to your work as a whole and that the electronic version is essentially equivalent to, or substitutes for, the print version.
5. A licence for 'post on a website' is valid for 12 months from the licence date. This licence does not cover use of full text articles on websites.
6. Where 'reuse in a dissertation/thesis' has been selected the following terms apply: Print rights of the final author's accepted manuscript (for clarity, NOT the published version) for up to 100 copies, electronic rights for use only on a personal website or institutional repository as defined by the Sherpa guideline (www.sherpa.ac.uk/romeo/).
7. Permission granted for books and journals is granted for the lifetime of the first edition and does not apply to second and subsequent editions

(except where the first edition permission was granted free of charge or for signatories to the STM Permissions Guidelines

http://www.stm-assoc.org/copyright-legal-affairs/permissions/permissions-guidelines/),

and does not apply for editions in other languages unless additional translation rights have been granted separately in the licence.

8. Rights for additional components such as custom editions and derivatives require additional permission and may be subject to an additional fee. Please apply to Journalpermissions@springernature.com/ bookpermissions@springernature.com for these rights.

9. The Licensor's permission must be acknowledged next to the licensed material in print. In electronic form, this acknowledgement must be visible at the same time as the figures/tables/illustrations or abstract, and must be hyperlinked to the journal/book's homepage. Our required acknowledgement format is in the Appendix below.

10. Use of the material for incidental promotional use, minor editing privileges (this does not include cropping, adapting, omitting material or any other changes that affect the meaning, intention or moral rights of the author) and copies for the disabled are permitted under this licence.

11. Minor adaptations of single figures (changes of format, colour and style) do not require the Licensor's approval. However, the adaptation should be credited as shown in Appendix below.

[...]
Version 1.1

In the excerpt above, the appendix templates are omitted. The applicable reference is Johanssen *et al.* [155].

Note that while Johanssen *et al.* [160] was published by Springer as well, it is licensed under Creative Commons as detailed in the next section.

A.5 Creative Commons

Work of two creative common licenses was used in this dissertation.

CC0 1.0 Universal (CC0 1.0) Public Domain Dedication

The work [152, 153] was published on CEUR-Ws.org. Use of this work is based on the Creative Commons CC0 license[43]—an excerpt is included in the following:

> **No Copyright**
>
> The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law.
>
> You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. [...]

Attribution 4.0 International (CC BY 4.0)

The work [160] was published on Springer Online. Use of this work is based on the Creative Commons Attribution 4.0 International License[44]—the respective excerpt:

> **You are free to:**
>
> **Share** — copy and redistribute the material in any medium or format
> **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.
>
> The licensor cannot revoke these freedoms as long as you follow the license terms.
>
> **Under the following terms:**
>
> **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
> **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[43] http://creativecommons.org/publicdomain/zero/1.0/.
[44] https://creativecommons.org/licenses/by/4.0.

Appendix B

Supplemental Investigation Material

In the following, we provide supplemental material for the case study described in Section 3.3. This includes a letter to address practitioners, as well as three slides that were provided via email.

B.1 Letter

Subject: Interview Request on the Subject of Continuous Software Engineering

[Salutation]

[optional personal Introduction]

Continuous Software Engineering (CSE) has changed the way software is being developed. Still, there are further, promising aspects of CSE left for research. For this purpose, we are working together with [UHD/TUM] on the research Project CURES ("Continuous Usage- and Rational-based Evolution Decision Support") with is supported by the German Research Foundation.

CURES aims for extending current CSE principles through innovative methods for user understanding and decision documentation. To align these to developers' and users' needs, we are conducting interviews with companies that apply CSE.

The central research questions are:

1. How do companies apply CSE during software evolution?
2. How do companies manage decisions and rationale during CSE?
3. How do companies involve users during CSE?

Therefore, we need an interview partner who worked in at least one project that applied CSE aspects; for example, a developer, project manager, or head of department. Attached, please find additional information regarding the interview as well as a mind map with topics regarding CSE for further orientation.

The interview will last about 90 minutes and can be conducted either through a personal meeting or on the phone. The interview will be conducted in English or German and will be recorded for evaluation purposes. Hereafter, we will provide the minutes to prevent misunderstandings. If you are interested, we will provide you with the interview results.

We plan to publish the evaluated information from the interviews and guarantee the anonymity of both the interview partner and the company.

I would appreciate to interview you or one of your colleague—please feel free to forward this mail including its attachment. In addition, I am happy to answer any of your questions, either via mail [mail] or phone [phone].

Best regards,

[Name]
[UHD/TUM]

B.2 Introduction Slide

Figure B.1 summarizes the relation between CSE and knowledge management.

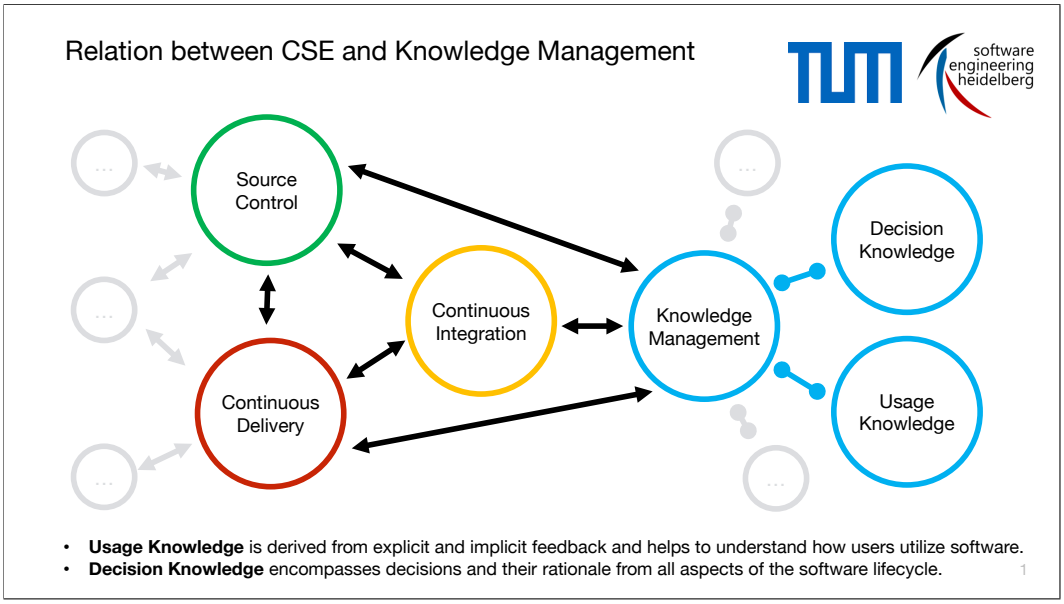


Figure B.1: Introduction Slide.

B.3 Requirements Slide

Figure B.1 describes requirements for participation in the interview.

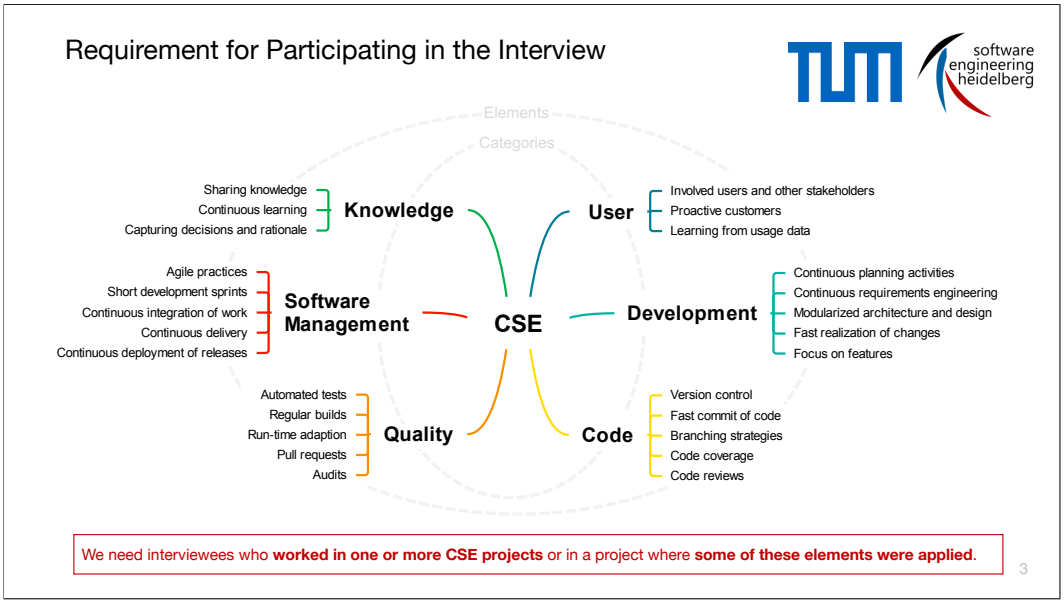


Figure B.2: Requirements Slide.

B.4 Consent Slide

Figure B.3 depicts the consent for participating in the interview.

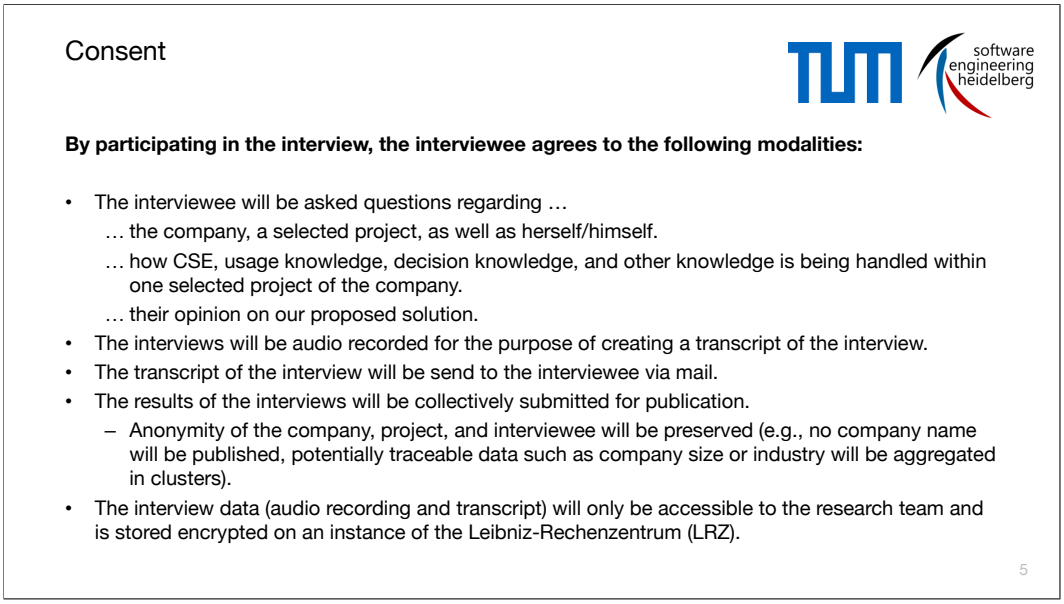


Figure B.3: Consent Slide.

Appendix C

Supplemental Treatment Material

This appendix provides addition figures to support the description of the **Cuu** SDK in Chapter 8 and the **Cuu** workflow in Chapter 9.

C.1 Start Screen

Figure C.1 depicts information screen that is presented to users in case they start an application that includes the **Cuu** SDK.

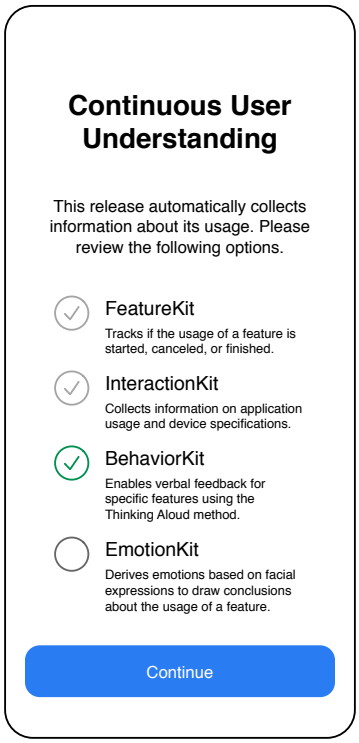


Figure C.1: The start screen of the **Cuu** SDK.

C.2 Spot Widgets

Figure C.2 depicts as FeatureKit spot widget that displays a list of detected feature crumbs (Ⓐ) and the number of observations (Ⓑ), i. e., how often they occurred.

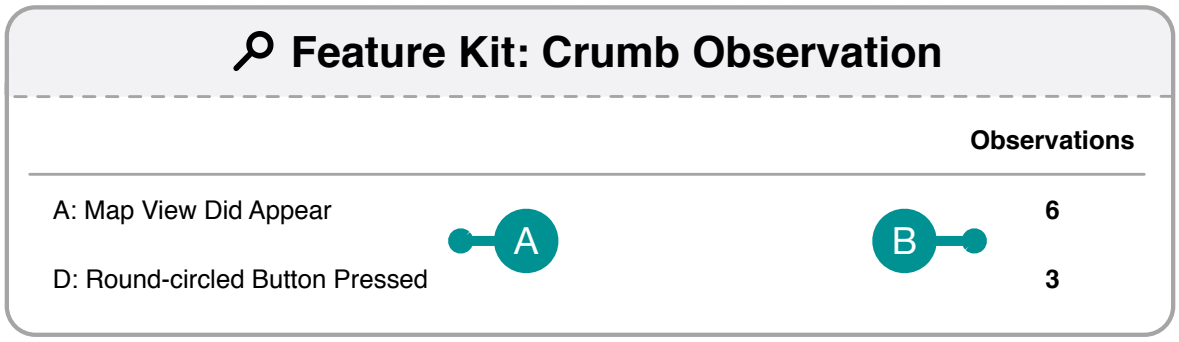


Figure C.2: Sketch of a spot widget for FeatureKit usage data.

Figure C.3 depicts as FeatureKit spot widget that displays a table of the number of observed feature states (Ⓑ) across observed devices types Ⓐ.

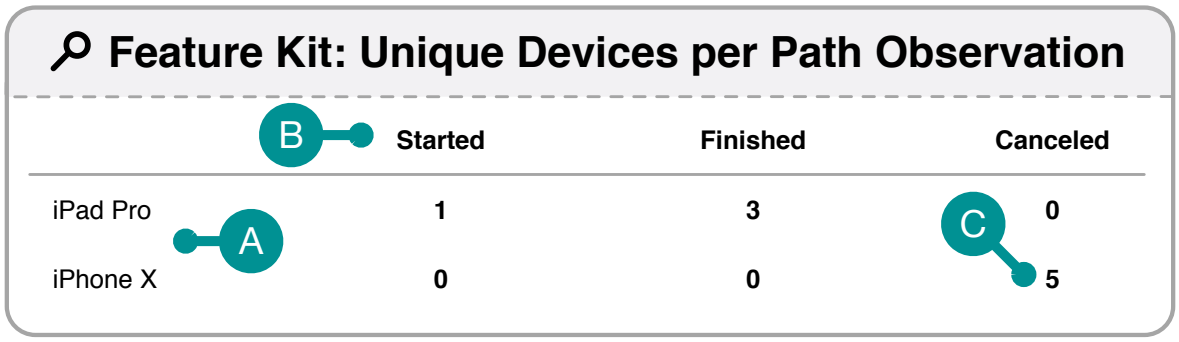


Figure C.3: Sketch of a spot widget for FeatureKit usage knowledge by devices.

Figure C.4 depicts as NoteKit spot widget that displays a list explicit user feedback with the respective timestamp.

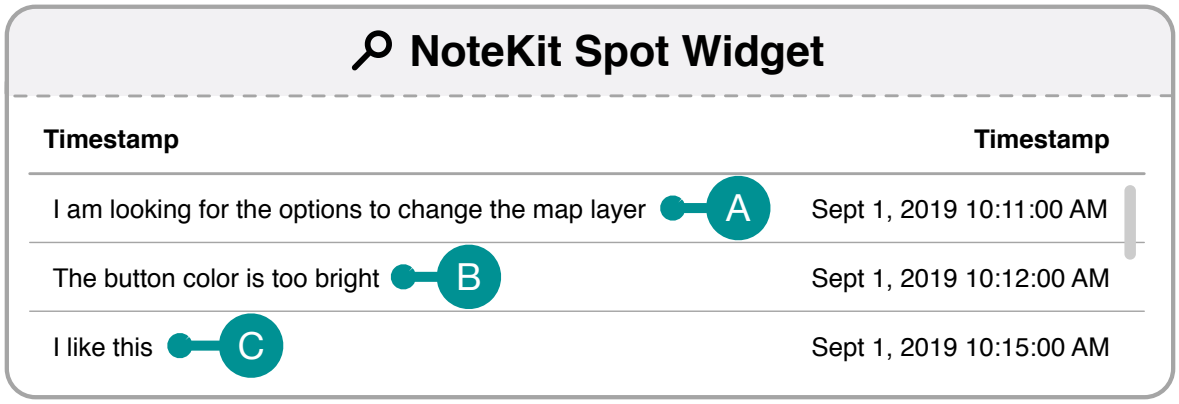


Figure C.4: Sketch of a spot widget for NoteKit for explicit user feedback.

C.3 EmotionKit

In Table C.1, we summarize the translation between the action units of the Facial Action Coding system (FACS) and ARKit results as described in Section 8.2.1.

Table C.1: Translation between FACS action units [90] and ARKit results.

AU	FACS name	BlendShapeLocations
1	Inner brow raiser	`.browInnerUp`
2	Outer brow raiser	`max(.browOuterUpLeft, .browOuterUpRight)`
4	Brow lowerer	`max(.browDownLeft, .browDownRight)`
5	Upper lid raiser	`max(.eyeLookUpLeft, .eyeLookUpRight)`
6	Cheek raiser	`max(.cheekSquintLeft, .cheekSquintRight)`
7	Lid tightener	`max(.eyeSquintLeft, .eyeSquintRight)`
8	Lips toward each other	`(.jawOpen + .mouthClose) * 0.5`
9	Nose wrinkler	`max(.noseSneerLeft, .noseSneerRight)`
10	Upper lip raiser	`(max(.mouthUpperUpLeft, .mouthUpperUpRight) + .mouthShrugUpper) * 0.5`
11	Nasolabial deepener	`(max(.mouthDimpleLeft, .mouthDimpleRight) + .mouthShrugUpper) * 0.5`
12	Lip corner puller	`max(.mouthSmileLeft, .mouthSmileRight)`
13	Sharp lip puller	`.mouthShrugLower`
14	Dimpler	`max(.mouthDimpleLeft, .mouthDimpleRight)`
15	Lip corner depressor	`max(.mouthFrownLeft, .mouthFrownRight)`
16	Lower lip depressor	`max(.mouthLowerDownLeft, .mouthLowerDownRight)`
18	Lip pucker	`.mouthPucker`
20	Lip stretcher	`max(.mouthStretchLeft, .mouthStretchRight)`
22	Lip funneler	`.mouthFunnel`
23	Lip tightener	`max(.mouthRollLower, .mouthRollUpper)`
24	Lip pressor	`max(.mouthPressLeft, .mouthPressRight)`
26	Jaw drop	`.jawOpen`

C.4 Workflow

Figure C.5 depicts a detailed version of Figure 9.6.

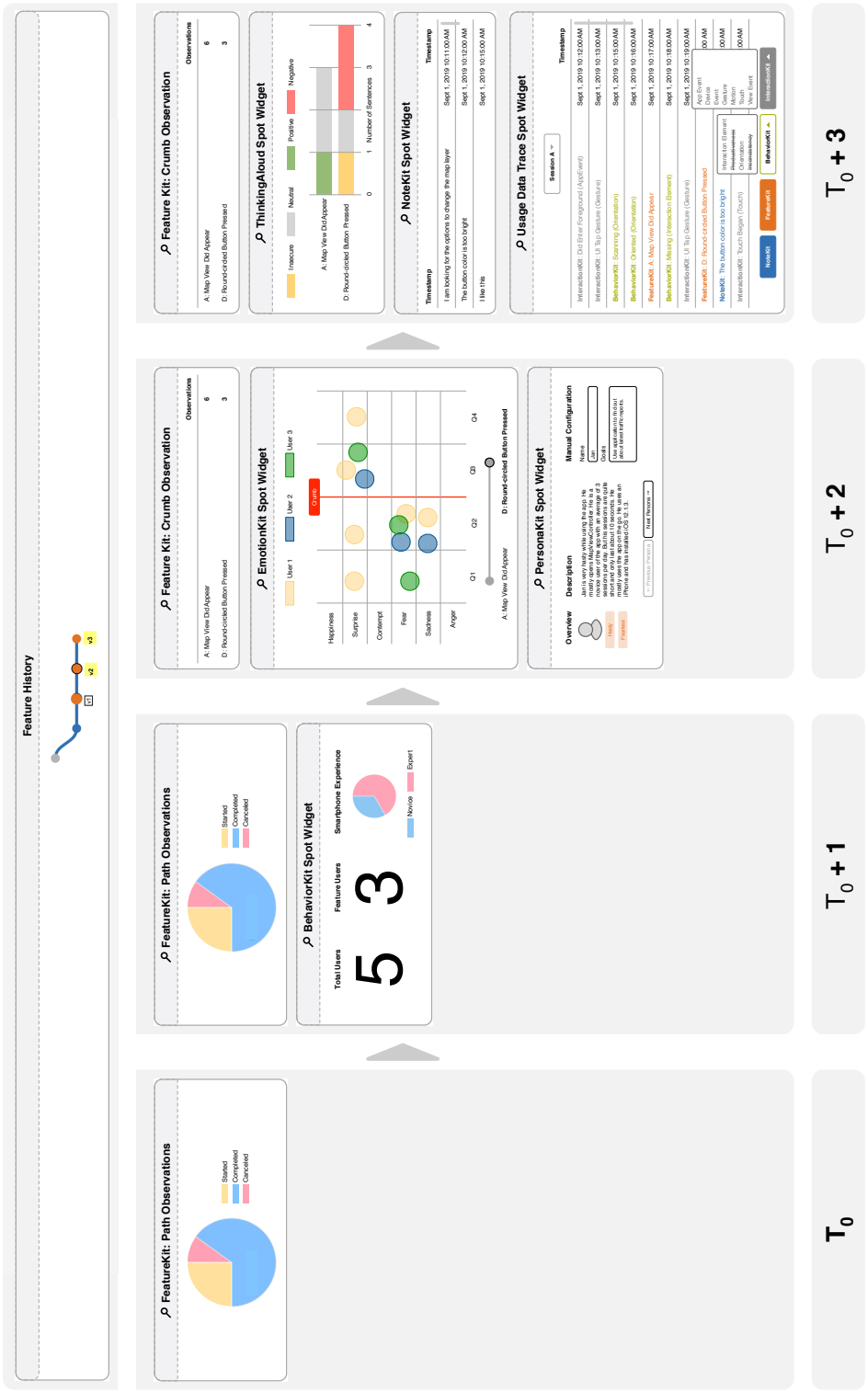


Figure C.5: Detailed change of the **Cuu** dashboard configurations over time.

Appendix D

Supplemental Validation Material

First, in Appendices D.1, D.2, and D.3, we present supplemental material for the validation of EmotionKit as presented in Chapter 10. Second, we introduce the two questionnaires that we used to validate the **Cuu**^{SE} framework following the TAM model in Chapter 12.

In Appendix D.4, we depict the first questionnaire containing four pages:

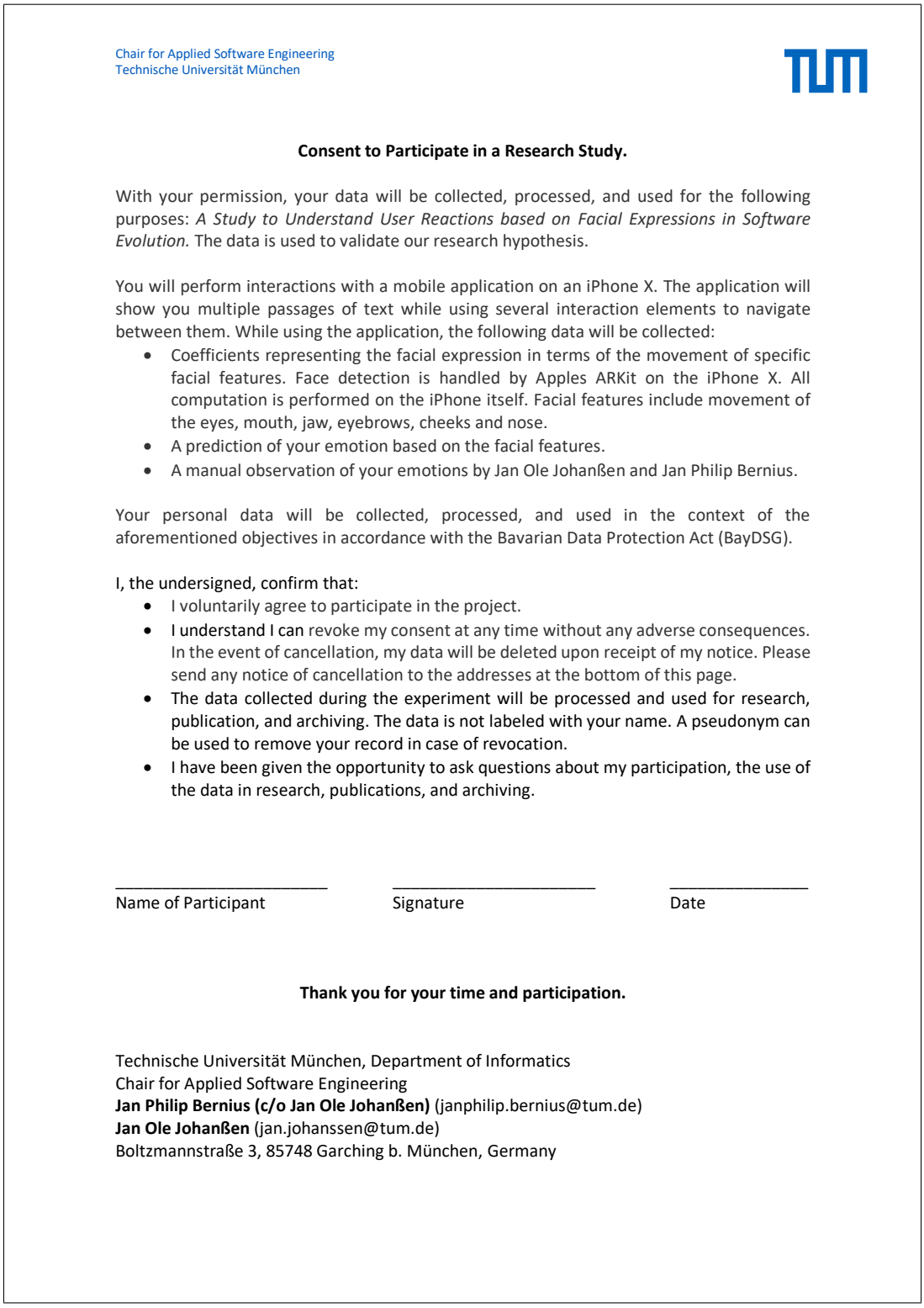
- Page 1 (Figure D.4) addresses concepts of usability engineering.
- Page 2 (Figure D.5) addresses the setup of **Cuu**^{SE}.
- Page 3 (Figure D.6) addresses the work with **Cuu**^{SE}.
- Page 4 (Figure D.7) addresses the understanding of **Cuu**^{SE}.

In Appendix D.5, we depict the second questionnaire containing 8 pages:

- Page 1 (Figure D.8) provides an introduction to a scenario to which the interviewees should relate their answers to.
- Page 2 (Figure D.9) addresses FeatureKit widgets.
- Page 3 (Figure D.10) addresses ThinkingAloudKit widgets.
- Page 4 (Figure D.11) addresses EmotionKit widgets.
- Page 5 (Figure D.12) addresses PersonaKit widgets.
- Page 6 addresses additional widgets that are not further referred to in this dissertation and were therefore omitted.
- Page 7 (Figure D.13) and page 8 (Figure D.14) address the future use of **Cuu**^{SE}.

D.1 Consent for Experiment

Figure D.1 depicts the consent which we used during the laboratory experiment.



Chair for Applied Software Engineering
Technische Universität München

Consent to Participate in a Research Study.

With your permission, your data will be collected, processed, and used for the following purposes: *A Study to Understand User Reactions based on Facial Expressions in Software Evolution.* The data is used to validate our research hypothesis.

You will perform interactions with a mobile application on an iPhone X. The application will show you multiple passages of text while using several interaction elements to navigate between them. While using the application, the following data will be collected:

- Coefficients representing the facial expression in terms of the movement of specific facial features. Face detection is handled by Apples ARKit on the iPhone X. All computation is performed on the iPhone itself. Facial features include movement of the eyes, mouth, jaw, eyebrows, cheeks and nose.
- A prediction of your emotion based on the facial features.
- A manual observation of your emotions by Jan Ole Johanßen and Jan Philip Bernius.

Your personal data will be collected, processed, and used in the context of the aforementioned objectives in accordance with the Bavarian Data Protection Act (BayDSG).

I, the undersigned, confirm that:

- I voluntarily agree to participate in the project.
- I understand I can revoke my consent at any time without any adverse consequences. In the event of cancellation, my data will be deleted upon receipt of my notice. Please send any notice of cancellation to the addresses at the bottom of this page.
- The data collected during the experiment will be processed and used for research, publication, and archiving. The data is not labeled with your name. A pseudonym can be used to remove your record in case of revocation.
- I have been given the opportunity to ask questions about my participation, the use of the data in research, publications, and archiving.

_____ _____ _____
Name of Participant Signature Date

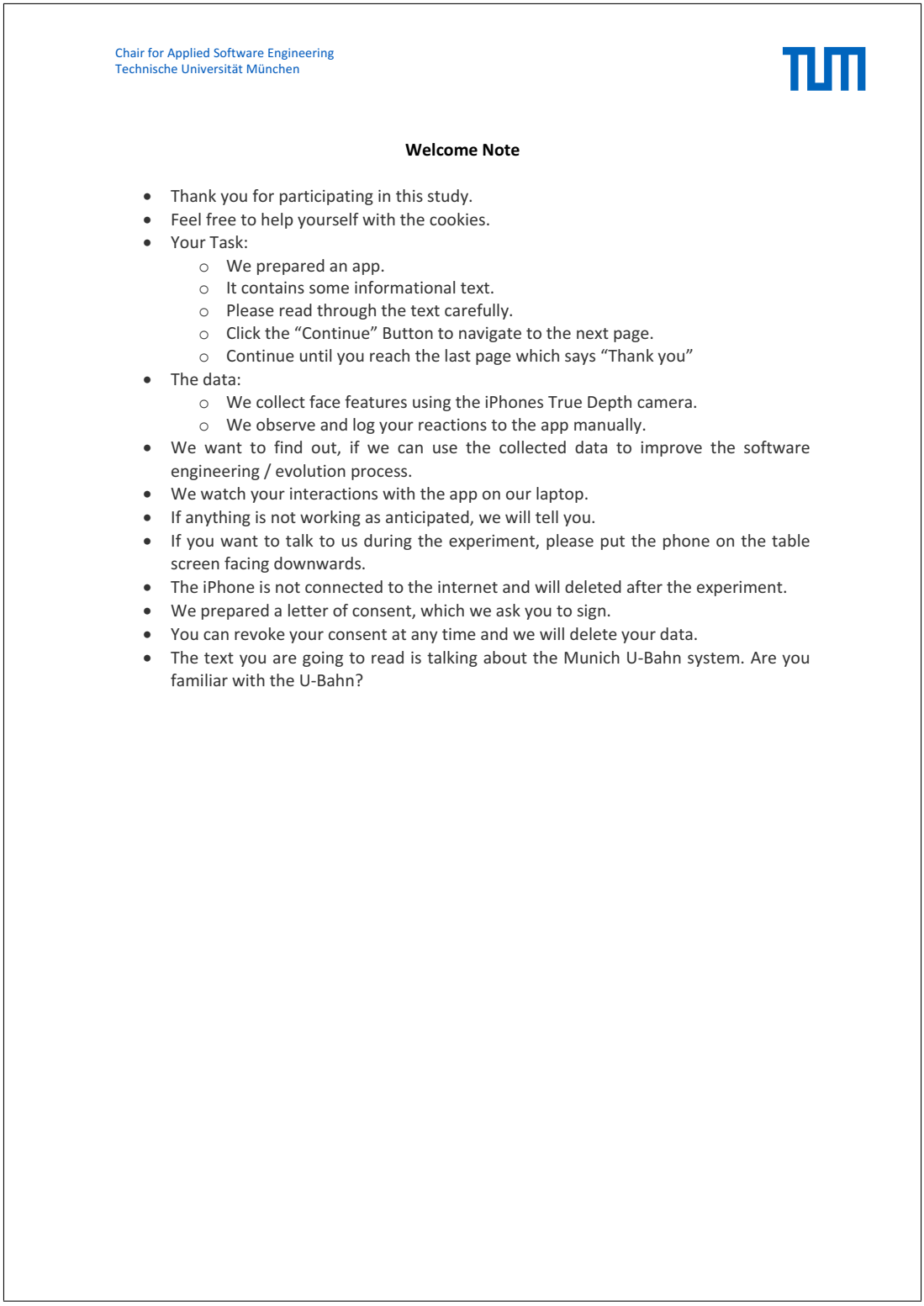
Thank you for your time and participation.

Technische Universität München, Department of Informatics
Chair for Applied Software Engineering
Jan Philip Bernius (c/o Jan Ole Johanßen) (janphilip.bernius@tum.de)
Jan Ole Johanßen (jan.johanssen@tum.de)
Boltzmannstraße 3, 85748 Garching b. München, Germany

Figure D.1: The consent used for the validation of EmotionKit.

D.2 Introduction Notes for Experiment

Figure D.2 depicts the notes which we used during the laboratory experiment.



Chair for Applied Software Engineering
Technische Universität München

Welcome Note

- Thank you for participating in this study.
- Feel free to help yourself with the cookies.
- Your Task:
  - We prepared an app.
  - It contains some informational text.
  - Please read through the text carefully.
  - Click the “Continue” Button to navigate to the next page.
  - Continue until you reach the last page which says “Thank you”
- The data:
  - We collect face features using the iPhones True Depth camera.
  - We observe and log your reactions to the app manually.
- We want to find out, if we can use the collected data to improve the software engineering / evolution process.
- We watch your interactions with the app on our laptop.
- If anything is not working as anticipated, we will tell you.
- If you want to talk to us during the experiment, please put the phone on the table screen facing downwards.
- The iPhone is not connected to the internet and will deleted after the experiment.
- We prepared a letter of consent, which we ask you to sign.
- You can revoke your consent at any time and we will delete your data.
- The text you are going to read is talking about the Munich U-Bahn system. Are you familiar with the U-Bahn?

Figure D.2: The introduction notes used for the validation of EmotionKit.

D.3 Observation Sheet for Experiment

Figure D.3 depicts the template which we used during the laboratory experiment.

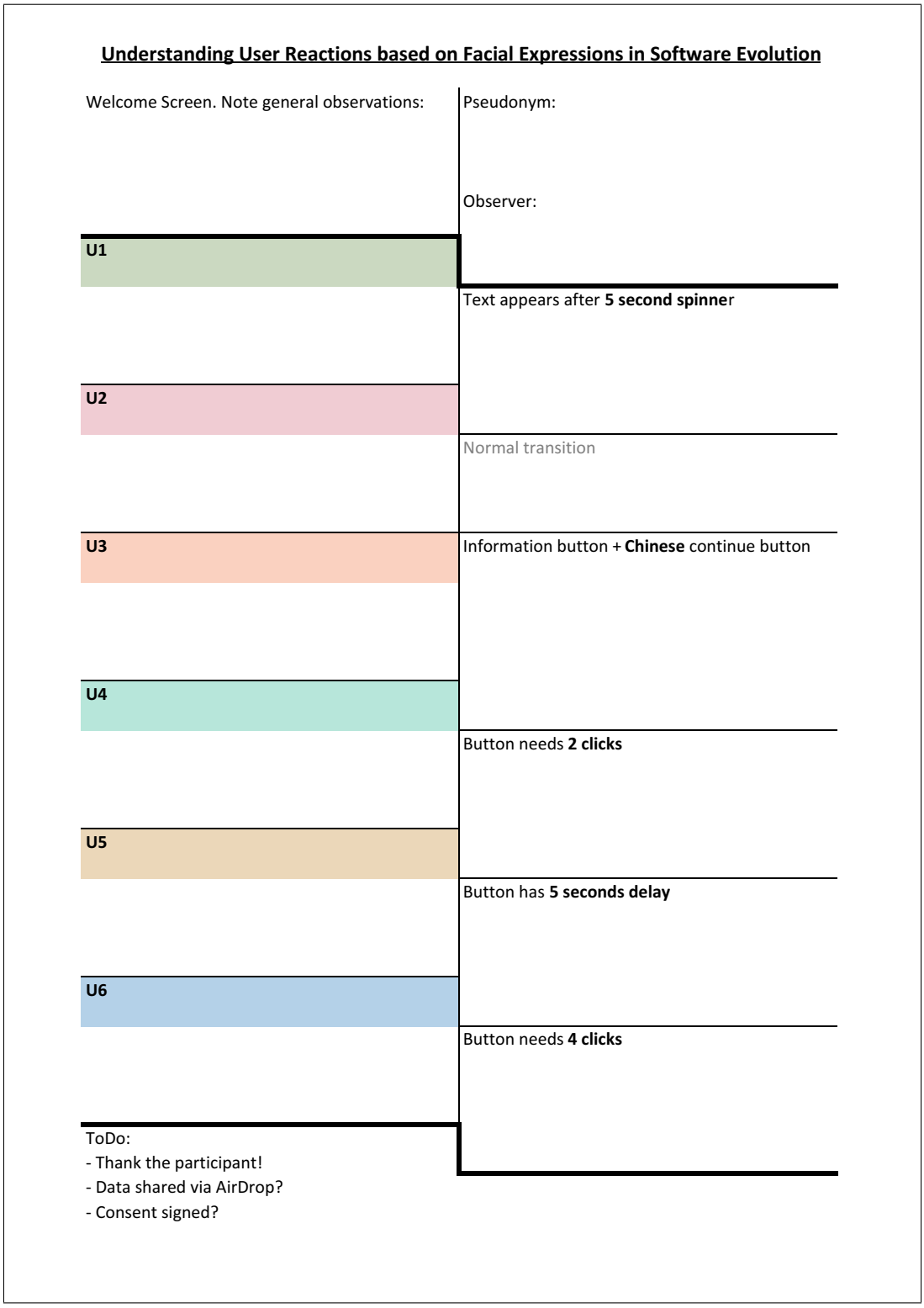


Figure D.3: The observation sheet used for the validation of EmotionKit.

D.4 First Questionnaire for Survey

Usability Engineering: Cross-Project Team iPraktikum WS18/19

Team Name	Current Sprint Number	Date

Dear Usability Manager, thank you for taking the time to fill in this questionnaire! *A few notes beforehand:*

- Answer from the **viewpoint of a developer** who is focused on usability engineering.
- Always **justify your answer** in the text box that follows the 🔍, e.g., by providing examples.
- Read the ℹ️ to get more **context** about a question.

Q1: Concepts of Usability Engineering

Q1.1 ► A usability engineering platform is useful to establish a common understanding of a feature.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 How can such a platform contribute to a feature understanding? What are better ways to create a common feature understanding? What is your team's feature understanding? What are differences or reasons for not having a common understanding of a feature?

Q1.2 ► Feature branches are useful to collect usage data for a feature.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think feature branches are (not) suitable for collecting usage data? If you disagree, what other options would you prefer?

Q1.3 ► The flow of events of a scenario is useful to analyze the usability of a related feature.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think they are helpful? If not, what would be an important source of information?

Usability Manager Questionnaire 1 / 4

Figure D.4: Page 1 of questionnaire 1.

Usability Engineering: Cross-Project Team iPraktikum WS18/19

Q2: Setting up CuuSE

Q2.1 ► Enough information to set up CuuSE was provided. ℹ See the 3rd UE Meeting.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 If there was enough information, what was particularly helpful? If not, what information was missing?

Q2.2 ► Linking CuuSE with Bitbucket is easy. ℹ See the webhook in Bitbucket.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think it was easy? If you disagree, at what step were you experiencing difficulties? How could the process be improved or simplified?

Q2.3 ► Initializing the CUU SDK within the Xcode project is easy.

ℹ *Initializing* refers to integrating the CUU pod, creating the `plist` file that includes `CUUTrackingToken` and `CUUProjectID`, and adding the `CUU.start()` and `CUU.stop()` methods in your iOS project.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 What were easy steps during the initialization? If you think it was not easy, at what step were you experiencing difficulties?

Usability Manager Questionnaire 2 / 4

Figure D.5: Page 2 of questionnaire 1.

Usability Engineering: Cross-Project Team iPraktikum WS18/19

Q3: Working with CuuSE

Q3.1 ► Creating a feature in the CuuSE platform is easy.

ℹ️ A CuuSE feature can be created for any new feature branch that was created based on a JIRA issue.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think it is easy? Should this process be improved? Do you have any suggestions for enhancements of the feature management in CuuSE?

Q3.2 ► Adding feature crumbs in the source code is easy.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think it is easy? If you disagree, how could this process be improved?

Q3.3 ► Creating a feature path in the CuuSE web interface is easy.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think it is easy? If you disagree, how could this process be improved?

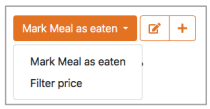
Figure D.6: Page 3 of questionnaire 1.

Q4: Understanding CuuSE

Q4.1 ► Switching between the usage data for different features is easy.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why do you think it is easy? If you disagree, how could this process be improved?



Q4.2 ► A linear feature path is useful to describe our feature under development.

ℹ️ CuuSE allows to only create feature paths that rely on a strict sequence of events.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 Why were linear paths particularly useful? If you disagree, should feature paths allow for modelling more complex feature executions? If yes, please provide an example of how you would use it.

Q4.3 ► The relationship between the git graph **springboard** and the usage data **widget** in the CuuSE platform is easy to understand.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

🔍 What makes the relationship easy to understand? If you disagree, can you give suggestions on how to clearly indicate that usage data of a selected commit is currently being shown in a widget?

Figure D.7: Page 4 of questionnaire 1.

D.5 Second Questionnaire for Survey

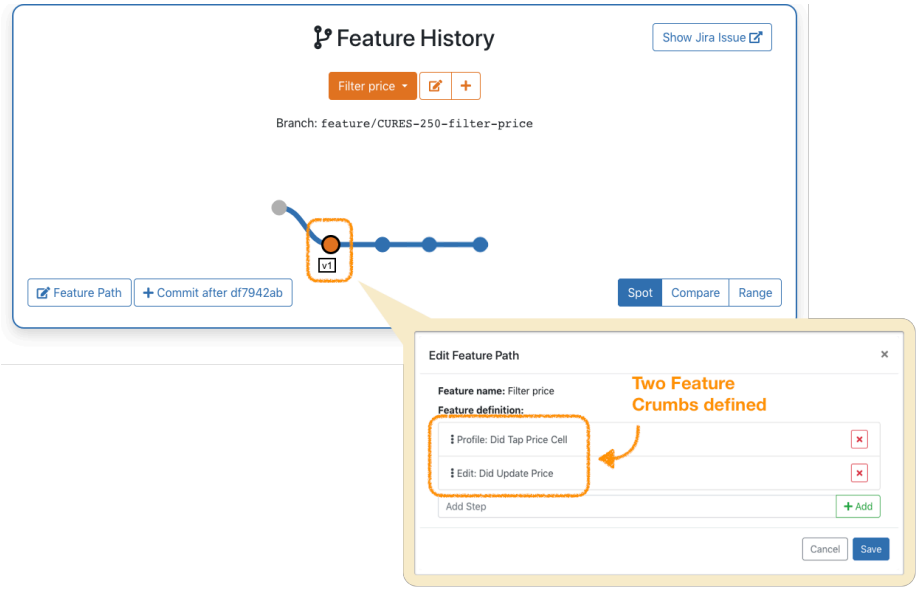
Cross-Project Team: Usability Engineering iPraktikum WS18/19

Team Name	Current Sprint Number	Date

Dear Usability Manager,

thank you for taking the time to fill in this questionnaire!

Please imagine the following scenario: You recently released a commit from a feature branch to your users that adds a new feature increment to your lunch application. This feature contains two feature crumbs: **Profile: Did Tap Price Cell** and **Edit: Did Update Price**.



In the following, we outline multiple CuuSE widgets that show usage knowledge that relates to both: the released commit and the two feature crumbs. We are interested in your thoughts about the widgets' expressiveness. In addition, we have some more general questions that go beyond CuuSE widgets.

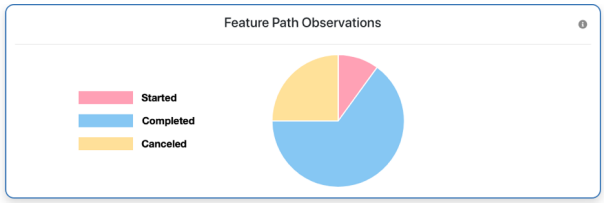
Please answer all questions from the **viewpoint of a developer** who is focused on usability engineering and try to **justify your answers**, e.g., by providing examples.

Usability Manager Questionnaire 1 / 8

Figure D.8: Page 1 of questionnaire 2.

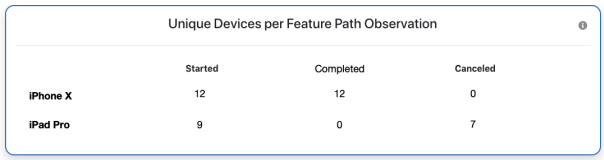
Q1: Feature Crumbs Widgets

Q1.1 ► Visualizing feature path observations separated by their state (*Started*, *Completed*, or *Canceled*) is useful to understand the feature's usability.



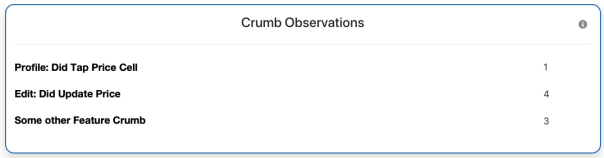
☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Q1.2 ► Visualizing the states of the feature path observations for unique device types is useful to understand the feature's usability.



☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Q1.3 ► Visualizing the number of feature crumb observations per individual feature crumb is useful to understand the feature's usability.



☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Q1.4 ► Why did you (dis)agree with the questions **Q1.1** to **Q1.3**? How can the widgets shown above be useful to improve the usability of your application? How could the widgets be improved? What other widget visualizations for feature crumbs can you think of?

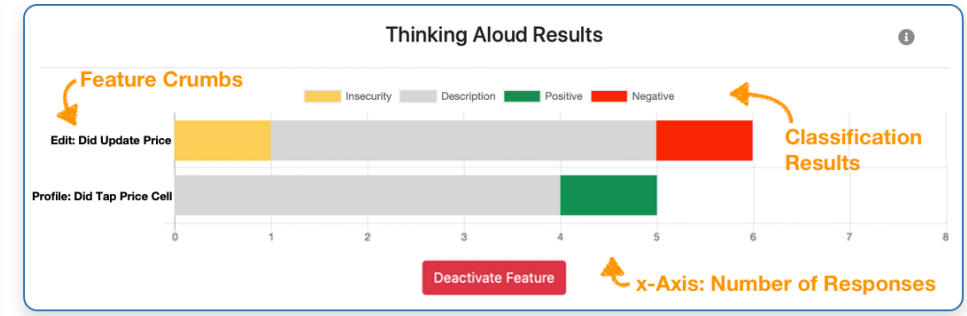
Figure D.9: Page 2 of questionnaire 2.

Q2: Thinking Aloud Widget

ℹ️ If **Continuous Thinking Aloud** is activated in CuuSE, the users will be asked to record verbal feedback as soon as they start using a new feature.

CuuSE automatically analyzes and classifies the responses into 🟨 **Insecurity** ("I don't know what to do"), ⬛ **Description** ("I see a button"), 🟩 **Positive** ("I like this button") and 🟥 **Negative** ("This is not good") feedback. The number of the classified responses is visualized per feature crumb.

Q2.1 ► Visualizing Thinking Aloud data, i.e., the number of classified verbal feedback, in relationship to feature crumbs is useful to understand the feature's usability.



☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Q2.2 ► Why do you (dis)agree with questions **Q2.1**? In your opinion, how could the Thinking Aloud widget be improved?

Q2.3 ► Provide a (visionary) example of how the Thinking Aloud widget and its results could be used to improve <u>your</u> application.

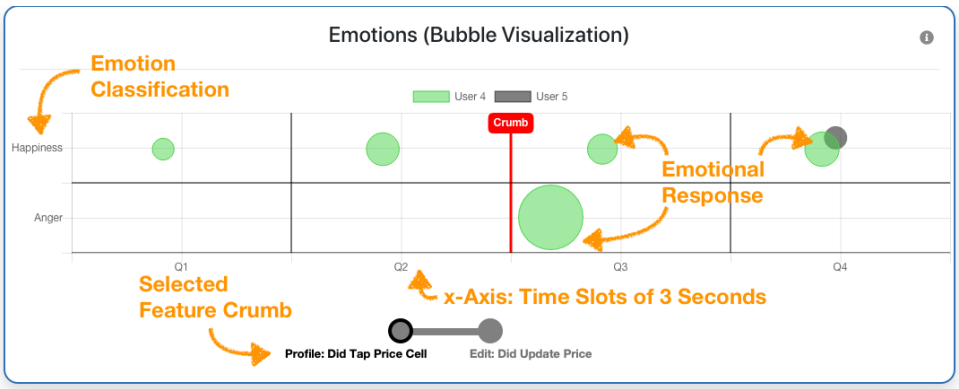
Figure D.10: Page 3 of questionnaire 2.

Q3: Emotion Widget

CuuSE can continuously observe a user to derive their emotional response towards an application. Therefore, a classifier predicts the user's emotions, such as **Happiness**, **Surprise**, or **Anger**.

CuuSE visualizes this information in relation to a feature crumb, divided into two quadrants of 3 seconds before and after the feature crumb was triggered. The intensity of a response is measured through the radius of a circle: *small radius → minor response* and *large radius → major response*.

Q3.1 ► Visualizing the emotional response, i.e., the users' state such as happiness or anger, in relationship to feature crumbs is useful to understand the feature's usability.



☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Q3.2 ► Why do you (dis)agree with **Q3.1**? In your opinion, how could the Emotion widget be improved?

Q3.3 ► Provide a (visionary) example of how the Emotion widget and its results could be used to improve your application.

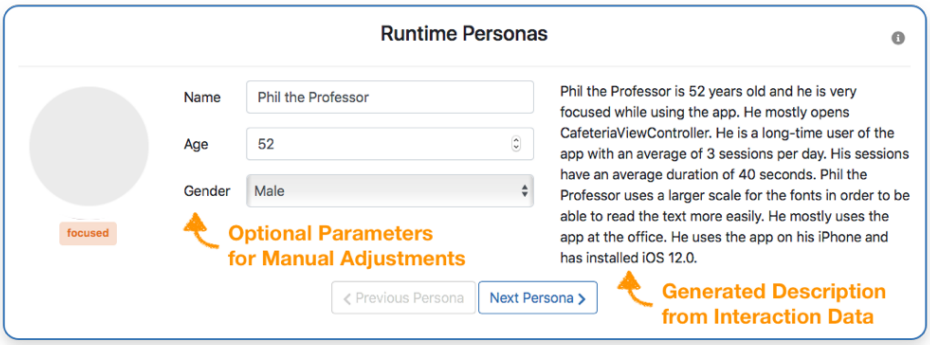
Figure D.11: Page 4 of questionnaire 2.

Q4: Runtime Persona Widget

ℹ️ Personas describe a fictional, potential user of an application with the goal of describing and better understanding the needs of a larger user group.

Based on a variety of interaction data, such as taps or time spans, CuuSE creates so-called **Runtime Personas** for every released commit. They contain textual descriptions of user characteristics.

Q4.1 ► Visualizing Runtime Personas for a release is useful to better fit the feature under development to users' requirements.



☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Q4.2 ► Why do you (dis)agree with questions **Q4.1**? In your opinion, how could the Runtime Persona widget be improved?

Q4.3 ► Provide a (visionary) example of how the Runtime Persona widget and its results could be used to improve your application.

Figure D.12: Page 5 of questionnaire 2.

Q7: Future Use of CuuSE

ℹ️ Please answer the following questions based on both: a) your previous, hands-on experience with CuuSE over the last weeks as well as b) the descriptions of widgets from above.

Q7.1 ► What other features do you expect for CuuSE that are not implemented or described above?

Q7.2 ► What other widgets for CuuSE would be useful to you? Why and for what would they be needed?

Q7.3 ► Besides usability engineering, can you think of other use cases for CuuSE? What other knowledge sources would be worth presenting in widgets?

Q7.4 ► I would add feature crumbs to my applications' source code to enable feature path observation visualization as shown in the widgets from **Q1**.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Figure D.13: Page 7 of questionnaire 2.

Q7.5 ► I would add, modify, or extend existing classes in my applications' source code to enable additional widgets in CuuSE, such as the Emotion widget shown in **Q3**.

For example, in order to have emotional data recorded by CuuSE, it is required to use the dedicated CUUViewController class instead of the standard UIViewController class.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Why do you (dis)agree?

Q7.6 ► CuuSE supports teams during usability engineering.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Why do you (dis)agree? What could be done to improve the usability engineering support of CuuSE?

Q7.7 ► I would use CuuSE in future projects.

☐ Strongly Agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly Disagree

Why do you (dis)agree?

Figure D.14: Page 8 of questionnaire 2.

List of Figures

List of Tables

Bibliography

[1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *1st International Symposium on Handheld and Ubiquitous Computing*, HUC, pages 304–307. Springer-Verlag, 1999.

[2] Russell L. Ackoff. From data to wisdom. *Journal of Applied Systems Analysis*, 16(1):3–9, 1989.

[3] Silvia T. Acuña, John W. Castro, and Natalia Juristo. A HCI technique for improving requirements elicitation. *Information and Software Technology*, 54(12):1357 – 1375, 2012.

[4] Enrique M. Albornoz and Diego H. Milone. Emotion recognition in never-seen languages using a novel ensemble method with emotion profiles. *IEEE Transactions on Affective Computing*, 8(1):43–53, January 2017.

[5] Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.

[6] Zoya Alexeeva, Diego Perez-Palacin, and Raffaela Mirandola. Design decision documentation: A literature overview. In *Software Architecture*, volume 5292 of *LNCS*, pages 84–101. Springer Berlin Heidelberg, 2016.

[7] Rana Alkadhi, Emitza Guzman, Jan O. Johanssen, and Bernd Bruegge. REACT: An Approach for Capturing Rationale in Chat Messages. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 175–180. ACM/IEEE, November 2017.

[8] Malik Almaliki, Cornelius Ncube, and Raian Ali. The design of adaptive acquisition of users feedback: An empirical study. In *8th International Conference on Research Challenges in Information Science*, RCIS, pages 1–12, May 2014.

[9] Lukas Alperowitz. *ProCeeD*. Dissertation, Technical University of Munich, München, 2017.

[10] Lukas Alperowitz, Dora Dzvonyar, and Bernd Bruegge. Metrics in agile project courses. In *38th International Conference on Software Engineering Companion*, ICSE, pages 323–326. ACM, 2016.

[11] Lukas Alperowitz, Jan O. Johanssen, Dora Dzvonyar, and Bernd Bruegge. Modeling in agile project courses. In *13th Educator Symposium*, MODELS (Satellite Events), pages 521–524, September 2017.

[12] Lukas Alperowitz, Andrea Marie Weintraud, Stefan Christoph Kofler, and Bernd Bruegge. Continuous prototyping. In *3rd International Workshop on Rapid Continuous Softwoftware Engineering*, pages 36–42. IEEE, 2017.

[13] Iosif Alvertis, Dimitris Papaspyros, Sotiris Koussouris, Spyros Mouzakitis, and Dimitris Askounis. Using crowdsourced and anonymized personas in the requirements elicitation and software development phases of software engineering. In *11th International Conference on Availability, Reliability and Security*, ARES, pages 851–856. IEEE, August 2016.

[14] Jisun An, Hoyoun Cho, Haewoon Kwak, Mohammed Z. Hassen, and Bernard J. Jansen. Towards automatic persona generation using social media. In *4th International Conference on Future Internet of Things and Cloud Workshops*, FiCloudW, pages 206–211. IEEE, August 2016.

[15] David J. Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[16] Nicolas Anquetil, Káthia M. de Oliveira, Kleiber D. de Sousa, and Márcio G. Batista Dias. Software maintenance seen as a knowledge management issue. *Information and Software Technology*, 49(5):515–529, 2007.

[17] Florian Auer and Michael Felderer. Current state of research on continuous experimentation: A systematic mapping study. In *44th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA, pages 335–344, August 2018.

[18] Hajer Ayed, Benoit Vanderose, and Naji Habra. Agile cultural challenges in europe and asia: insights from practitioners. In *39th International Conference on Software Engineering*, ICSE, pages 153–162. IEEE, 2017.

[19] Earl R. Babbie. *Survey Research Methods*. Wadsworth, 1990.

[20] Daniel Bader and Dennis Pagano. Towards automated detection of mobile usability issues. In *1st European Workshop on Mobile Engineering*, ME, pages 341–354, 2013.

[21] Elsa Bakiu and Emitza Guzman. Which feature is unusable? detecting usability and user experience issues from user reviews. In *25th International Requirements Engineering Conference Workshops*, REW, pages 182–187. IEEE, September 2017.

[22] Tadas Baltrusaitis, Amir Zadeh, Yao Chong Lim, and Louis-Philippe Morency. Openface 2.0: Facial behavior analysis toolkit. In *13th International Conference on Automatic Face Gesture Recognition*, FG, pages 59–66. IEEE, May 2018.

[23] Muneera Bano and Didar Zowghi. Users' involvement in requirements engineering and system success. In *3rd International Workshop on Empirical Requirements Engineering*, EmpiRE, pages 24–31, July 2013.

[24] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. The bones of the system: A case study of logging and telemetry at microsoft. In *International Conference on Software Engineering*, pages 92–101. IEEE, 2016.

[25] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *18th International Conference on Software Engineering*, ICSE, pages 442–449. IEEE, March 1996.

[26] Victor R. Basili, Caldiera Gianluigi, and Dieter H. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 1994.

[27] Victor R. Basili and H. Dieter Rombach. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

[28] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[29] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[30] Andrew Begel. Invited talk: Fun with software developers and biometrics. In *1st International Workshop on Emotional Awareness in Software Engineering*, SEmotion, pages 1–2. IEEE/ACM, May 2016.

[31] Izak Benbasat, David K. Goldstein, and Melissa Mead. The case research strategy in studies of information systems. *MIS Quarterly*, 11(3):369–386, 1987.

[32] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature?:

A qualitative study of features in industrial software product lines. In *19th International Conference on Software Product Line*, SPLC, pages 16–25. ACM, 2015.

[33] Jan P. Bernius. Detecting and visualizing user emotions in software evolution. Master's thesis, Technical University of Munich, October 2018.

[34] Jan P. Bernius. Understanding user reactions based on facial expressions in software evolution. Guided research, Technical University of Munich, March 2018.

[35] Jane Billestrup, Jan Stage, Anders Bruun, Lene Nielsen, and Kira S. Nielsen. Creating and using personas in software development: Experiences from practice. In Stefan Sauer, Cristian Bogdan, Peter Forbrig, Regina Bernhaupt, and Marco Winckler, editors, *Human-Centered Software Engineering*, pages 251–258. Springer Berlin Heidelberg, 2014.

[36] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[37] Barry Boehm. The future of software and systems engineering processes. *University of Southern California, Los Angeles, CA*, 2005.

[38] Barry Boehm, Alexander Egyed, Dan Port, Archita Shah, Julie Kwan, and Ray Madachy. A stakeholder win-win approach to software engineering education. *Annals of Software Engineering*, 6(1/4):295–321, 1998.

[39] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI, pages 47–56. ACM, 2011.

[40] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *1st Edition of the Workshop on Mobile Cloud Computing*, MCC, pages 13–16. ACM, 2012.

[41] Jan Bosch. *Continuous Software Engineering: An Introduction*. Springer, 2014.

[42] Lionel C. Briand, Christiane M. Differding, and H. Dieter Rombach. Practical guidelines for measurement-based process improvement. *Software Process: Improvement and Practice*, 2(4):253–280, 1996.

[43] Jonalan Brickey, Steven Walczak, and Tony Burgess. Comparing semi-automated clustering methods for persona development. *IEEE Transactions on Software Engineering*, 38(3):537–546, May 2012.

[44] William H. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1998.

[45] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall Press, 3rd edition, 2010.

[46] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education*, 15(4):17:1–17:31, 2015.

[47] João Brunet, Gail C. Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. Do developers discuss design? In *11th Working Conference on Mining Software Repositories*, MSR, pages 340–343. ACM, 2014.

[48] Anders Bruun, Peter Gull, Lene Hofmeister, and Jan Stage. Let your users do the testing: a comparison of three remote asynchronous usability testing methods. In *Conference on Human Factors in Computing Systems*, CHI, pages 1619–1628. ACM, 2009.

[49] Anders Bruun, Effie Lai-Chong Law, Matthias Heintz, and Poul Svante Eriksen. Asserting real-time emotions through cued-recall: Is it valid? In *9th Nordic Conference on Human-Computer Interaction*, pages 37:1–37:10. ACM, 2016.

[50] Anders Bruun and Jan Stage. Barefoot usability evaluations. *Behaviour & Information Technology*, 33(11):1148–1167, 2014.

[51] Janis A. Bubenko. Challenges in requirements engineering. In *International Symposium on Requirements Engineering*, RE, pages 160–162, March 1995.

[52] Frank Buschmann, Kevin Henney, and Douglas C Schmidt. *Pattern-oriented software architecture, on patterns and pattern languages*, volume 1. John Wiley & Sons, 1996.

[53] Leydi Caballero, Ana M. Moreno, and Ahmed Seffah. Persona as a tool to involving human in agile methods: Contributions from hci and marketing. In Stefan Sauer, Cristian Bogdan, Peter Forbrig, Regina Bernhaupt, and Marco Winckler, editors, *Human-Centered Software Engineering*, pages 283–290. Springer Berlin Heidelberg, 2014.

[54] Michel Cabanac. What is emotion? *Behavioural Processes*, 60(2):69–83, November 2002.

[55] Laura V. Galvis Carreño and Kristina Winbladh. Analysis of user comments: An approach for software requirements evolution. In *35th International Conference on Software Engineering*, ICSE, pages 582–591, May 2013.

[56] John M. Carroll. *Scenario-based design: envisioning work and technology in system development*. Wiley, 1995.

[57] John M. Carroll and Mary B. Rosson. A case library for teaching usability engineering: Design rationale, development, and classroom experience. *Journal on Educational Resources in Computing*, 5(1):3, 2005.

[58] Kapil Chalil Madathil and Joel S. Greenstein. Synchronous remote usability testing: A new approach facilitated by virtual worlds. In *Conference on Human Factors in Computing Systems*, CHI, pages 2225–2234. ACM, 2011.

[59] Susy S. Chan, Rosalee J. Wolfe, and Xiaowen Fang. Issues and strategies for integrating hci in masters level mis and e-commerce programs. *International Journal of Human-Computer Studies*, 59(4):497 – 520, 2003.

[60] Lianping Chen. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72 – 86, 2017.

[61] Parmit Chilana, Christina Holsberry, Flavio Oliveira, and Andrew Ko. Designing for a billion users: A case study of facebook. In *Extended Abstracts on Human Factors in Computing Systems*, CHI EA, pages 419–432. ACM, 2012.

[62] Jürgen Cito, Fábio Oliveira, Philipp Leitner, Priya Nagpurkar, and Harald Gall. Context-based analytics: establishing explicit links between runtime traces and source code. In *International Conference on Software Engineering*, ICSE, pages 193–202. IEEE, 2017.

[63] Sue A. Conger. *The New Software Engineering*. International Thomson Publishing, 1st edition, 1994.

[64] Alan Cooper. *The Inmates Are Running the Asylum*. Macmillan Publishing Co., Inc., 1999.

[65] David Coppit and Jennifer M. Haddox-Schatz. Large team projects in software engineering courses. In *36th SIGCSE Technical Symposium on Computer Science Education*, pages 137–141. ACM, 2005.

[66] Daniel Corkill. Blackboard Systems. *AI Expert*, 6(9), January 1991.

[67] Iain D. Craig. Blackboard systems. *Artificial Intelligence Review*, 2(2):103–118, June 1988.

[68] Lisa Crispin. Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23(6):70–71, 2006.

[69] Bill Curtis. Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68(9):1144–1157, September 1980.

[70] Kostadin Damevski, David C. Shepherd, Johannes Schneider, and Lori Pollock. Mining sequences of developer interactions in visual studio for usage smells. *IEEE Transactions on Software Engineering*, 43(4):359–371, April 2017.

[71] Charles Darwin. *The Expression of Emotion in Man and Animals.* Project Gutenberg, 1872.

[72] Alan M. Davis. *Software Requirements: Objects, Functions, and States*. Prentice-Hall, Inc., 1993.

[73] Fred D. Davis. *A technology acceptance model for empirically testing new end-user information systems: Theory and results*. PhD thesis, Massachusetts Institute of Technology, 1985.

[74] Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989.

[75] Fred D. Davis, Richard P. Bagozzi, and Paul R. Warshaw. User Acceptance of Computer Technology: A Comparison of Two Theoretical Models. *Management Science*, 35(8):982 – 1002, 1989.

[76] Fred D. Davis and Viswanath Venkatesh. Toward preprototype user acceptance testing of new information systems: implications for software project management. *IEEE Transactions on Engineering Management*, 51(1):31–46, February 2004.

[77] John Dawes. Do data characteristics change according to the number of scale points used? an experiment using 5-point, 7-point and 10-point scales. *International Journal of Market Research*, 50(1):61–104, 2008.

[78] Marco de Sá and Luís Carriço. Designing and evaluating mobile interaction: Challenges and trends. *Foundations and Trends in Human–Computer Interaction*, 4(3):175–243, March 2011.

[79] Anind K. Dey. Enabling the use of context in interactive applications. In *Extended Abstracts on Human Factors in Computing Systems*, CHI EA, pages 79–80. ACM, 2000.

[80] Jeremy Dick, Elizabeth Hull, and Ken Jackson. Introduction. In *Requirements engineering*. Springer, 2017.

[81] Edsger W. Dijkstra. On webster, users, bugs and aristotle. 1977.

[82] Edsger W. Dijkstra. On webster, users, bugs and aristotle. In *Selected Writings on Computing: A Personal Perspective*, pages 288–291. Springer-Verlag, 1982.

[83] Trinh M. T. Do, Jan Blom, and Daniel Gatica-Perez. Smartphone usage in the wild: A large-scale analysis of applications and context. In *13th International Conference on Multimodal Interfaces*, ICMI, pages 353–360. ACM, 2011.

[84] Yuxiao Dong, Yang Yang, Jie Tang, Yang Yang, and Nitesh V. Chawla. Inferring user demographics and social strategies in mobile social networks. In *20th International Conference on Knowledge Discovery and Data Mining*, pages 15–24. ACM, 2014.

[85] Rafael Durán-Sáez, Xavier Ferré, Hongming Zhu, and Qin Liu. Task analysis-based user event logging for mobile applications. In *25th International Requirements Engineering Conference Workshops*, REW, pages 152–155. IEEE, September 2017.

[86] Allen H. Dutoit, Ray McCall, Ivan Mistrík, and Barbara Paech, editors. *Rationale Management in Software Engineering*. Springer-Verlag, 2006.

[87] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9):833–859, 2008.

[88] Dora Dzvonyar, Stephan Krusche, Rana Alkadhi, and Bernd Bruegge. Context-aware user feedback in continuous software evolution. In *International Workshop on Continuous Software Evolution and Delivery*, CSED, pages 12–18. ACM, 2016.

[89] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. *Selecting Empirical Methods for Software Engineering Research*, chapter 11, pages 285–311. Springer London, London, 2008.

[90] Paul Ekman and Wallace V. Friesen. *Facial Action Coding System*. Number 1. Consulting Psychologists Press, 1978.

[91] Paul Ekman, Wallace V. Friesen, and Phoebe Ellsworth. *Emotion in the Human Face: Guidelines for Research and an Integration of Findings (General Psychology)*. Pergamon Press, 1972.

[92] Paul Ekman, Wallace V. Friesen, Phoebe Ellsworth, Arnold P. Goldstein, and Leonard Krasner. *Emotion in the Human Face: Guidelines for Research and an Integration of Findings*. Pergamon general psychology series. Elsevier Science, 2013.

[93] Stefan Elsen. Visgi: Visualizing git branches. In *1st Working Conference on Software Visualization*, VISSOFT, pages 1–4. IEEE, September 2013.

[94] Thomas Erickson. Notes on design practice: Stories and prototypes as catalysts for communication. In John M. Carroll, editor, *Scenario-based Design*, pages 37–58. John Wiley & Sons, Inc., 1995.

[95] Fabian Fagerholm, Alejandro Guinea, Hanna Mäenpää, and Jürgen Münch. Building blocks for continuous experimentation. In *1st International Workshop on Rapid Continuous Software Engineering*, pages 26–35. ACM, 2014.

[96] Fabian Fagerholm, Alejandro Guinea, Hanna Mäenpää, and Jürgen Münch. The right model for continuous experimentation. *Journal of Systems and Software*, 123:292 – 305, 2017.

[97] Shamal Faily and Ivan Flechais. Persona cases: A technique for grounding personas. In *Conference on Human Factors in Computing Systems*, CHI, pages 2267–2270. ACM, 2011.

[98] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23(1):452–489, Feb 2018.

[99] Jackson Feijó Filho, Wilson Prata, and Juan Oliveira. Affective-ready, contextual and automated usability test for mobile software. In *18th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct*, MobileHCI, pages 638–644. ACM, 2016.

[100] Jackson Feijó Filho, Wilson Prata, and Juan Oliveira. Where-how-what am i feeling: User context logging in automated usability tests for mobile software. In Aaron Marcus, editor, *Design, User Experience, and Usability: Technological Contexts*, pages 14–23, Cham, 2016. Springer International Publishing.

[101] Jackson Feijó Filho, Thiago Valle, and Wilson Prata. Automated usability tests for mobile devices through live emotions logging. In *17th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct*, MobileHCI, pages 636–643. ACM, 2015.

[102] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, July 2013.

[103] Norman Fenton, Shari L. Pfleeger, and Robert L. Glass. Science and substance: a challenge to software engineers. *IEEE Software*, 11(4):86–95, July 1994.

[104] Xavier Ferre, Elena Villalba, Héctor Julio, and Hongming Zhu. Extending mobile app analytics for usability test logging. In *Human-Computer Interaction*, INTERACT, pages 114–131. Springer, 2017.

[105] Bruna Moraes Ferreira, Simone D. J. Barbosa, and Tayana Conte. Pathy: Using empathy with personas to design applications that meet the users' needs. In Masaaki Kurosu, editor, *Human-Computer Interaction. Theory, Design, Development and Practice*, pages 153–165, Cham, 2016. Springer International Publishing.

[106] Arlene Fink. *The Survey Handbook*. SAGE Publications, 2003.

[107] Arlene Fink. *How to design survey studies*. SAGE Publications, 2010.

[108] Florian Fittschen. Automatic generation of personas from usage data during continuous software engineering. Master's thesis, Technical University of Munich, October 2018.

[109] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[110] Ilias Flaounas and Arik Friedman. Bridging the gap between business, design and product metrics. In *Extended Abstracts of the Conference on Human Factors in Computing Systems*, CHI EA. ACM, 2019.

[111] Alexandra Fountaine and Bonita Sharif. Emotional awareness in software development: Theory and measurement. In *2nd International Workshop on Emotion Awareness in Software Engineering*, SEmotion, pages 28–31. IEEE/ACM, May 2017.

[112] Xavier Franch, Claudia Ayala, Lidia López, Silverio Martínez-Fernández, Pilar Rodríguez, Cristina Gómez, Andreas Jedlitschka, Markku Oivo, Jari Partanen, Timo Räty, and Veikko Rytivaara. Data-driven requirements engineering in agile projects: The q-rapids approach. In *25th International Requirements Engineering Conference Workshops*, REW, pages 411–414. IEEE, 2017.

[113] Lex Fridman, Daniel E. Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Aleksandr Patsekin, Julia Kindelsberger, Li Ding, Sean Seaman, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie Seppelt, Linda Angell, Bruce Mehler, and Bryan Reimer. MIT advanced vehicle technology study: Large-scale naturalistic driving study of driver behavior and interaction with automation. *IEEE Access*, 7:102021–102038, 2019.

[114] Michael Fröhlich. User classification based on behavior patterns in mobile applications. Master's thesis, Technical University of Munich, February 2018.

[115] Davide Fucci, Cristina Palomares, Xavier Franch, Dolors Costal, Mikko Raatikainen, Martin Stettinger, Zijad Kurtanovic, Tero Kojo, Lars Koenig, Andreas Falkner, Gottfried Schenner, Fabrizio Brasca, Tomi Männistö, Alexander Felfernig, and Walid Maalej. Needs and challenges for a platform to support large-scale requirements engineering: A multiple-case study. In *12th International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 19:1–19:10. ACM, 2018.

[116] Michael J. Gallivan and Mark Keil. The user–developer communication process: a critical case study. *Information Systems Journal*, 13(1):37–68, 1 2003.

[117] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[118] Vincenzo Gervasi, Ricardo Gacitua, Mark Rouncefield, Peter Sawyer, Leonid Kof, Lin Ma, Paul Piwek, Anne De Roeck, Alistair Willis, Hui Yang, and Bashar Nuseibeh. *Unpacking Tacit Knowledge for Requirements Engineering*, pages 23–47. Springer Berlin Heidelberg, 2013.

[119] Carlo Ghezzi and Dino Mandrioli. The challenges of software engineering education. In Paola Inverardi and Mehdi Jazayeri, editors, *International Conference on Software Engineering*, ICSE, pages 115–127. Springer Berlin Heidelberg, 2006.

[120] Gerd Gigerenzer. *Bauchentscheidungen: Die Intelligenz des Unbewussten und die Macht der Intuition*. Goldmann Verlag, 2008.

[121] María Gómez, Bram Adams, Walid Maalej, Martin Monperrus, and Romain Rouvoy. App store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems. *IEEE Software*, 34(2):81–89, Mar 2017.

[122] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, 1994.

[123] John C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, 27(4):857–871, 1971.

[124] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc., 1992.

[125] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, Inc., 1987.

[126] Peggy Gregory, Leonor Barroca, Helen Sharp, Advait Deshpande, and Katie Taylor. The challenges that challenge: Engaging with agile practitioners' concerns. *Information and Software Technology*, 77:92 – 104, 2016.

[127] Eduard C. Groen, Sylwia Kopczyńska, Marc P. Hauer, Tobias D. Krafft, and Joerg Doerr. Users — the hidden software product quality experts?: A study on how app users report quality aspects in online reviews. In *25th International Requirements Engineering Conference*, RE, pages 80–89. IEEE, September 2017.

[128] Fredrik Gundelsweiler, Thomas Memmel, and Harald Reiterer. Agile usability engineering. In *Mensch & computer*, pages 33–42, 2004.

[129] Thomas Günzel. App user behavior tracking and visualization. Bachelor's thesis, Technical University of Munich, October 2016.

[130] Emitza Guzman, Padma Bhuvanagiri, and Bernd Bruegge. Fave: Visualizing user feedback for software evolution. In *2nd Working Conference on Software Visualization*, pages 167–171. IEEE, September 2014.

[131] Emitza Guzman, Mohamed Ibrahim, and Martin Glinz. A little bird told me: Mining tweets for requirements and software evolution. In *25th International Requirements Engineering Conference*, RE, pages 11–20. IEEE, September 2017.

[132] Emitza Guzman, Mohamed Ibrahim, and Martin Glinz. Mining twitter messages for software evolution. In *39th International Conference on Software Engineering Companion*, ICSE, pages 283–284. IEEE/ACM, May 2017.

[133] Emitza Guzman and Walid Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *22nd International Requirements Engineering Conference*, RE, pages 153–162. IEEE, August 2014.

[134] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[135] Jean Hartley. Case study research. In Catherine Cassel and Gillian Symon, editors, *Essential guide to qualitative methods in organizational research*, pages 323–333. SAGE Publications, 2004.

[136] Regina Hebig. UI-Tracer: A Lightweight Approach to Help Developers Tracing User Interface Elements to Source Code. In *Software Engineering und Software Management 2018*, pages 225–236, 2018.

[137] Judith S. Heinisch, Christoph Anderson, and Klaus David. Angry or climbing stairs? towards physiological emotion recognition in the wild. In *International Conference on Pervasive Computing and Communications Workshops*, PerCom, pages 486–491. IEEE, March 2019.

[138] Juho Heiskari and Laura Lehtola. Investigating the State of User Involvement in Practice. In *6th Asia-Pacific Software Engineering Conference*, pages 433–440, Penang, Malaysia, 2009. IEEE.

[139] Nicholas L. Henry. Knowledge management: A new concern for public administration. *Public Administration Review*, 34(3):189–196, 1974.

[140] Morten Hertzum, Pia Borlund, and Kristina B. Kristoffersen. What do thinking-aloud participants say? a comparison of moderated and unmoderated usability sessions. *International Journal of Human–Computer Interaction*, 31(9):557–570, 2015.

[141] Tom-Michael Hesse, Arthur Kuehlwein, and Tobias Roehm. Decdoc: A tool for documenting design decisions collaboratively and incrementally. In *1st International Workshop on Decision Making in Software Architecture*, pages 30–37. IEEE, 2016.

[142] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, March 2004.

[143] Christopher K. Hobbs and Herbert H. Tsang. Industry in the classroom: Equipping students with real-world experience a reflection on the effects of industry partnered projects on computing education. In *Western Canadian Conference on Computing Education*, WCCCE, pages 1–5. ACM, 2014.

[144] Helena Holmström Olsson and Jan Bosch. Towards data-driven product development: A multiple case study on post-deployment data usage in software-intensive embedded systems. In Brian Fitzgerald, Kieran Conboy, Ken Power, Ricardo Valerdi, Lorraine Morgan, and Klaas-Jan Stol, editors, *Lecture Notes in Business Information Processing*, volume 167, pages 152–164. Springer Berlin Heidelberg, 2013.

[145] Andreas Holzinger. Usability engineering methods for software developers. *Communications of the ACM - Interaction design and children*, 48(1):71–74, January 2005.

[146] Watts S. Humphrey. The software engineering process: Definition and scope. In *4th International Software Process Workshop on Representing and Enacting the Software Process*, ISPW, pages 82–83. ACM, 1988.

[147] Melody Y. Ivory and Marti A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4):470–516, December 2001.

[148] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer London, London, 2008.

[149] Angus Jenkinson. Beyond segmentation. *Journal of targeting, measurement and analysis for marketing*, 3(1):60–72, 1994.

[150] Jan O. Johanssen. Continuous user understanding for the evolution of interactive systems. In *Symposium on Engineering Interactive Computing Systems*, EICS, pages 15:1–15:6. ACM, 2018.

[151] Jan O. Johanssen, Jan P. Bernius, and Bernd Bruegge. Toward usability problem identification based on user emotions derived from facial expressions. In *4th International Workshop on Emotion Awareness in Software Engineering*, SEmotion, pages 1–7. IEEE/ACM, May 2019.

[152] Jan O. Johanssen, Dominic Henze, and Bernd Bruegge. A syllabus for usability engineering in multi-project courses. In *16. Workshop Software Engineering im Unterricht der Hochschulen*, SEUH, pages 133–144, 2019.

[153] Jan O. Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. Towards a systematic approach to integrate usage and decision knowledge in continuous software engineering. In Stephan Krusche, Horst Lichter, Dirk Riehle, and Andreas Steffens, editors, *2nd Workshop on Continuous Software Engineering*, CSE, pages 7–11, February 2017.

[154] Jan O. Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. Towards the visualization of usage and decision knowledge in continuous software engineering. In *Working Conference on Software Visualization*, VISSOFT, pages 104–108. IEEE, September 2017.

[155] Jan O. Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. Feature crumbs: Adapting usage monitoring to continuous software engineering. In Marco Kuhrmann, Kurt Schneider, Dietmar Pfahl, Sousuke Amasaki, Marcus Ciolkowski, Regina Hebig, Paolo Tell, Jil Klünder, and Steffen Küpper, editors, *Product-Focused Software Process Improvement*, pages 263–271, Cham, 2018. Springer International Publishing.

[156] Jan O. Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. How do practitioners capture and utilize user feedback during continuous software

engineering? In *27th International Requirements Engineering Conference*, RE, pages 153–164. IEEE, September 2019.

[157] Jan O. Johanssen, Anja Kleebaum, Barbara Paech, and Bernd Bruegge. Practitioners' eye on continuous software engineering: An interview study. In *International Conference on Software and System Process*, ICSSP, pages 41–50. ACM, 2018.

[158] Jan O. Johanssen, Anja Kleebaum, Barbara Paech, and Bernd Bruegge. Continuous software engineering and its support by usage and decision knowledge: An interview study with practitioners. *Journal of Software: Evolution and Process*, 31(5):e2169, 2019.

[159] Jan O. Johanssen, Lara M. Reimer, and Bernd Bruegge. Continuous thinking aloud. In *Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution*, RCoSE-DDrEE, pages 12–15. IEEE/ACM, 2019.

[160] Jan O. Johanssen, Fabien P. Viertel, Bernd Bruegge, and Kurt Schneider. *Tacit Knowledge in Software Evolution*, pages 77–105. Springer International Publishing, Cham, 2019.

[161] Peter Johnson, Hilary Johnson, and Stephanie Wilson. Rapid prototyping of user interfaces driven by task models. In John M. Carroll, editor, *Scenario-based Design*, pages 209–246. John Wiley & Sons, Inc., 1995.

[162] Soon-Gyo Jung, Jisun An, Haewoon Kwak, Moeed Ahmad, Lene Nielsen, and Bernard J. Jansen. Persona generation from aggregated social media data. In *Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA, pages 1748–1755. ACM, 2017.

[163] Lena Karlsson, Åsa Dahlstedt, Johan Natt och Dag, Björn Regnell, and Anne Persson. Challenges in market-driven requirements engineering-an industrial interview study. In *8th International Workshop on Requirements Engineering: Foundation for Software Quality*, REFSQ, September 2002.

[164] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., 1990.

[165] Katja Kevic, Brendan Murphy, Laurie Williams, and Jennifer Beckmann. Characterizing experimentation in continuous deployment: A case study on bing. In *39th International Conference on Software Engineering*, ICSE, pages 123–132, May 2017.

[166] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[167] Anja Kleebaum, Jan O. Johanssen, Barbara Paech, Rana Alkadhi, and Bernd Bruegge. Decision knowledge triggers in continuous software engineering. In *4th International Workshop on Rapid Continuous Software Engineering*, RCoSE, pages 23–26. ACM, 2018.

[168] Anja Kleebaum, Jan O. Johanssen, Barbara Paech, and Bernd Bruegge. Tool support for decision and usage knowledge in continuous software engineering. In Stephan Krusche, Horst Lichter, Dirk Riehle, and Andreas Steffens, editors, *3rd Workshop on Continuous Software Engineering*, CSE, pages 74–77, March 2018.

[169] Anja Kleebaum, Jan O. Johanssen, Barbara Paech, and Bernd Bruegge. How do practitioners manage decision knowledge during continuous software engineering? In *31st International Conference on Software Engineering and Knowledge Engineering*, SEKE, pages 735–740. KSI Research Inc., 2019.

[170] Anja Kleebaum, Jan O. Johanssen, Barbara Paech, and Bernd Bruegge. Teaching rationale management in agile project courses. In *16. Workshop Software Engineering im Unterricht der Hochschulen*, SEUH, pages 125–132, 2019.

[171] Alessia Knauss. On the usage of context for requirements elicitation: End-user involvement in it ecosystems. In *20th International Requirements Engineering Conference*, RE, pages 345–348. IEEE, September 2012.

[172] Andrew J. Ko, Michael J. Lee, Valentina Ferrari, Steven Ip, and Charlie Tran. A case study of post-deployment user feedback triage. In *4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE, pages 1–8. ACM Press, 2011.

[173] Thomas Kosch, Markus Funk, Albrecht Schmidt, and Lewis L. Chuang. Identifying cognitive assistance with mobile electroencephalography: A case study with in-situ projections for manual assembly. *ACM Human-Computer Interaction*, 2(EICS):11:1–11:20, June 2018.

[174] Helmut Krcmar. *Informationsmanagement*. Springer, 2015.

[175] Stephan Krusche. *Rugby - A Process Model for Continuous Software Engineering*. Dissertation, Technical University of Munich, 2016.

[176] Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin Wagner. Rugby: An agile process model based on continuous delivery. In *1st International Workshop on Rapid Continuous Software Engineering*, RCoSE, pages 42–50. ACM, 2014.

[177] Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. Teaching code review management using branch based workflows. In *International Conference on Software Engineering - Companion Volume*, pages 384–393, 2016.

[178] Stephan Krusche and Bernd Bruegge. User feedback in mobile development. In *2nd International Workshop on Mobile Development Lifecycle*, MobileDeLi, pages 25–26. ACM, 2014.

[179] Stephan Krusche and Bernd Bruegge. CSEPM - a continuous software engineering process metamodel. 3rd International Workshop on Rapid Continuous Software Engineering, pages 2–8. IEEE/ACM, May 2017.

[180] Stephan Krusche, Nadine von Frankenberg, and Sami Afifi. Experiences of a software engineering course based on interactive learning. In *Software Engineering im Unterricht der Hochschulen*, SEUH, pages 32–40, 2017.

[181] Marco Kuhrmann, Philipp Diebold, Jürgen Münch, Paolo Tell, Vahid Garousi, Michael Felderer, Kitija Trektere, Fergal McCaffery, Oliver Linssen, Eckhart Hanser, and Christian R. Prause. Hybrid software and system development in practice: Waterfall, scrum, and beyond. In *International Conference on Software and System Process*, ICSSP, pages 30–39, Paris, France, 2017. ACM.

[182] Sari Kujala. User involvement: A review of the benefits and challenge. *Behaviour & Information Technology*, 22:1–16, 2003.

[183] Werner Kunz and Horst W. J. Rittel. Issues as elements of information systems. Working Paper No. 131, 1970.

[184] Marta Larusdottir, Jan Gulliksen, and Åsa Cajander. A license to kill – Improving UCSD in Agile development. *Journal of Systems and Software*, 123:214–222, 2017.

[185] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *28th International Conference on Software Engineering*, ICSE, pages 492–501. ACM, 2006.

[186] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, 82:55 – 79, 2017.

[187] Larix Lee and Philippe Kruchten. A tool to visualize architectural design decisions. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *International Conference on the Quality of Software Architectures*, pages 43–54. Springer, 2008.

[188] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[189] Luis Leiva. Automatic web design refinements based on collective user behavior. In *Extended Abstracts on Human Factors in Computing Systems*, CHI EA, pages 1607–1612. ACM, 2012.

[190] Victor R. Lesser, Richard D. Fennell, Lee D. Erman, and D. Raj Reddy. Organization of the hearsay ii speech understanding system. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(1):11–24, February 1975.

[191] Clayton Lewis, Peter G. Polson, Cathleen Wharton, and John Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Conference on Human Factors in Computing Systems*, CHI '90, pages 235–242. ACM, 1990.

[192] Michael Lewis, Jeannette M. Haviland-Jones, and Lisa Feldman Barrett, editors. *Handbook of Emotions*. The Guilford Press, 3rd edition edition, 2010.

[193] Stan Z. Li and Anil K. Jain, editors. *Handbook of Face Recognition*. Springer London, 2011.

[194] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22, 1932.

[195] Lin Liu, Qing Zhou, Jilei Liu, and Zhanqiang Cao. Requirements cybernetics: Elicitation based on user behavioral data. *Journal of Systems and Software*, 124:187 – 194, 2017.

[196] Walid Maalej, Hans-Jörg Happel, and Asarnusch Rashid. When users become collaborators: Towards continuous and context-aware user input. In *24th SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 981–990. ACM, 2009.

[197] Walid Maalej, Maleknaz Nayebi, Timo Johann, and Guenther Ruhe. Toward data-driven requirements engineering. *IEEE Software*, 33(1):48–54, January 2016.

[198] Simo Mäkinen, Marko Leppänen, Terhi Kilamo, Anna-Liisa Mattila, Eero Laukkanen, Max Pagels, and Tomi Männistö. Improving the delivery cycle: A

multiple-case study of the toolchains in finnish software intensive enterprises. *Information and Software Technology*, 80:175–194, 2016.

[199] Eric Malmi and Ingmar Weber. You are what apps you use: Demographic prediction based on user's apps. In *10th International Conference on Web and Social Media*, ICWSM, 2016.

[200] Regan L. Mandryk and M. Stella Atkins. A fuzzy physiological approach for continuously modeling emotion during interaction with play technologies. *International Journal of Human-Computer Studies*, 65(4):329 – 347, 2007.

[201] Nikola Marangunić and Andrina Granić. Technology acceptance model: A literature review from 1986 to 2013. *Universal Access in the Information Society*, 14(1):81–95, March 2015.

[202] Jevgeni Marenkov, Tarmo Robal, and Ahto Kalja. A study on immediate automatic usability evaluation of web application user interfaces. In Guntis Arnicans, Vineta Arnicane, Juris Borzovs, and Laila Niedrite, editors, *Databases and Information Systems*, pages 257–271, Cham, 2016. Springer International Publishing.

[203] Stephanie Larissa Marsh, Jason Dykes, and Fenia Attilakou. Evaluating a Geovisualization Prototype with Two approaches: Remote Instructional vs. Face-to-Face Exploratory. In *10th International Conference on Information Visualisation*, IV, pages 310–315, July 2006.

[204] Catherine Marshall and Gretchen B. Rossman. *Designing Qualitative Research*. SAGE Publications, 1995.

[205] Deborah Mayhew. *The usability engineering lifecycle: a practitioner's handbook for user interface design*. Morgan Kaufmann, 1999.

[206] Daniel McDuff, Rana El Kaliouby, Jeffrey F. Cohn, and Rosalind W. Picard. Predicting ad liking and purchase intent: Large-scale analysis of facial responses to ads. *IEEE Transactions on Affective Computing*, 6(3):223–235, July 2015.

[207] Patrick Mennig, Simon A. Scherr, and Frank Elberzhager. Supporting rapid product changes through emotional tracking. In *4th International Workshop on Emotion Awareness in Software Engineering*, SEmotion, pages 8–12. IEEE/ACM, May 2019.

[208] Mathias Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014.

[209] Matthew B. Miles and Alan M. Huberman. *Qualitative data analysis: An expanded sourcebook*. SAGE Publications, 1994.

[210] Thomas P. Moran. Guest editor's introduction: An applied psychology of the user. *ACM Computing Surveys*, 13(1):1–11, March 1981.

[211] Steve Mulder and Ziv Yaar. *The User is Always Right: A Practical Guide to Creating and Using Personas for the Web*. New Riders Publishing, 1st edition, 2006.

[212] Myriam Munezero, Calkin Suero Montero, Maxim Mozgovoy, and Erkki Sutinen. Emotwitter – a fine-grained visualization system for identifying enduring sentiments in tweets. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, pages 78–91, Cham, 2015. Springer International Publishing.

[213] Michael D. Myers and Michael Newman. The Qualitative Interview in IS Research: Examining the Craft. *Information and Organization*, 17(1):2–26, 2007.

[214] Allen Newell. Some problems of basic organization in problem-solving programs. In Marshal C. Yovits, George T. Jacobi, and Gordon D. Goldstein, editors, *Conference on Self-Organizing Systems*, May 1962.

[215] Allen Newell. Heuristic programming: Ill-structured problems. In Julius S. Aronofsky, editor, *Progress in Operations Research*, volume III, 1969.

[216] Jakob Nielsen. *Usability Engineering*. Interactive Technologies. Elsevier Science, 1994.

[217] Jakob Nielsen. Scenarios in discount usability engineering. In John M. Carroll, editor, *Scenario-based Design*, pages 59–83. John Wiley & Sons, Inc., 1995.

[218] Jakob Nielsen, Rita M. Bush, Tom Dayton, Nancy E. Mond, Michael J. Muller, and Robert W. Root. Teaching experienced developers to design graphical user interfaces. In *Conference on Human Factors in Computing Systems*, CHI, pages 557–564. ACM, 1992.

[219] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *Conference on Human Factors in Computing Systems*, CHI, pages 206–213. ACM, 1993.

[220] Jakob Nielsen and Rolf Molich. Teaching user interface design based on usability engineering. *SIGCHI Bulletin*, 21(1):45–48, August 1989.

[221] Penny H. Nii. Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53, 1986.

[222] Penny H. Nii. Blackboard systems, part two: Blackboard application systems, blackboard systems from a knowledge engineering perspective. *AI Magazine*, 7(3):82–106, 1986.

[223] Ikujiro Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, 1995.

[224] Ikujiro Nonaka and Hirotaka Takeuchi. *Die Organisation des Wissens: Wie japanische Unternehmen eine brachliegende Ressource nutzbar machen*. Campus Verlag, 2012.

[225] Donald A. Norman. *The Design of Future Things*. Basic books, 2007.

[226] Donald A. Norman. *The design of everyday things: Revised and expanded edition*. Basic Books, 2013.

[227] Donald A. Norman and Stephen W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., 1986.

[228] Tom Nurkkala and Stefan Brandle. Software studio: Teaching professional software engineering. In *Technical Symposium on Computer Science Education*, pages 153–158. ACM, 2011.

[229] Charlene O'Hanlon. A conversation with werner vogels. *Queue*, 4(4):14:14–14:22, May 2006.

[230] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the "Stairway to Heaven" - a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 392–399, September 2012.

[231] Marc Oriol, Melanie Stade, Farnaz Fotrousi, Sergi Nadal, Jovan Varga, Norbert Seyff, Alberto Abello, Xavier Franch, Jordi Marco, and Oleg Schmidt. Fame: Supporting continuous requirements elicitation by combining user feedback and monitoring. In *26th International Requirements Engineering Conference*, RE, pages 217–227. IEEE, August 2018.

[232] Tina Øvad, Nis Bornoe, Lars Bo Larsen, and Jan Stage. Teaching software developers to perform ux tasks. In *Annual Meeting of the Australian Special*

Interest Group for Computer Human Interaction, OzCHI, pages 397–406. ACM, 2015.

[233] Barbara Paech. Project memories: Integrating knowledge and requirements management. In *5th International Workshop on Requirements Engineering: Foundations for Software Quality*, REFSQ, pages 43–47, Heidelberg, Germany, 1999.

[234] Volker Paelke and Karsten Nebe. Integrating agile methods for mixed reality design space exploration. In *Conference on Designing interactive systems*, pages 240–249. ACM, 2008.

[235] Dennis Pagano. *Portneuf - A Framework for Continuous User Involvement*. Dissertation, Technical University of Munich, 2013.

[236] Dennis Pagano and Bernd Brügge. User involvement in software evolution practice: A case study. In *International Conference on Software Engineering*, ICSE, pages 953–962. IEEE, 2013.

[237] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *International Conference on Software Maintenance and Evolution*, ICSME, pages 281–290. IEEE, September 2015.

[238] Maja Pantic. *Facial Expression Recognition*. Springer US, 1 edition, 2009.

[239] Julia Paredes, Craig Anslow, and Frank Maurer. Information visualization for agile software development. In *2nd Working Conference on Software Visualization*, VISSOFT, pages 157–166. IEEE, 2014.

[240] Vimla L. Patel, Jose F. Arocha, and David R. Kaufman. Expertise and tacit knowledge in medicine. In *Tacit knowledge in professional practice: Researcher and Practitioner Perspectives*. Lawrence Erlbaum Associates, 1998.

[241] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models*, pages 362–369. Springer US, 1997.

[242] Antonio Pérez, M Isabel García, Manuel Nieto, José L Pedraza, Santiago Rodríguez, and Juan Zamorano. Argos: An advanced in-vehicle data recorder on a massively sensorized vehicle for car driver behavior experimentation. *IEEE Transactions on Intelligent Transportation Systems*, 11(2):463–473, June 2010.

[243] Gary Perlman. Teaching user interface development to software engineers. *Human Factors Society Annual Meeting*, 32(5):391–394, 1988.

[244] Gary Perlman. Teaching user interface development to software engineers. In *Conference Companion on Human Factors in Computing Systems*, CHI, pages 375–376. ACM, 1995.

[245] Shari Lawrence Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1(1):219–253, December 1995.

[246] Rosalind W. Picard. Affective computing. *M.I.T Media Laboratory Perceptual Computing Section Technical Report No. 321*, 1995.

[247] Klaus Pohl and Chris Rupp. *Requirements Engineering Fundamentals*. Rocky Nook, 2nd edition, 2015.

[248] Michael Polanyi. *The Tacit Dimension*. University of Chicago Press, 2009.

[249] Karl R. Popper. *Objective knowledge: An evolutionary approach*. Oxford University Press, 1972.

[250] John Pruitt and Jonathan Grudin. Personas: Practice and theory. In *Conference on Designing for User Experiences*, DUX, pages 1–15. ACM, 2003.

[251] Mona Rahimi and Jane Cleland-Huang. Personas in the middle: Automated support for creating personas as focal points in feature gathering forums. In *29th International Conference on Automated Software Engineering*, ASE, pages 479–484. ACM, 2014.

[252] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference*, AGILE, pages 1–10, August 2015.

[253] Balasubramaniam Ramesh, Lan Cao, and Richard Baskerville. Agile requirements engineering practices and challenges: an empirical study. *Information Systems Journal*, 20(5):449–480, 2010.

[254] Lara M. Reimer. Thinking aloud in continuous testing. Master's thesis, Technical University of Munich, October 2018.

[255] Horst W. J. Rittel and Melvin M. Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4(2):155–169, Jun 1973.

[256] Tarmo Robal, Jevgeni Marenkov, and Ahto Kalja. Ontology design for automatic evaluation of web user interface usability. In *Portland International Conference on Management of Engineering and Technology*, PICMET, pages 1–8, July 2017.

[257] Colin Robson. *Real world research*. Blackwell, 2nd edition, 2002.

[258] Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123:263–291, 2017.

[259] Tobias Röhm. *The MALTASE Framework For Usage-Aware Software Evolution*. Dissertation, Technical University of Munich, 2015.

[260] Rasmus Ros and Per Runeson. Continuous experimentation and a/b testing: A mapping study. In *4th International Workshop on Rapid Continuous Software Engineering*, pages 35–41. ACM, 2018.

[261] Mary Beth Rosson and John M. Carroll. Narrowing the specification-implementation gap in scenario-based design. In John M. Carroll, editor, *Scenario-based Design*, pages 247–278. John Wiley & Sons, Inc., 1995.

[262] William B. Rouse. Human-computer interaction in the control of dynamic systems. *ACM Computing Surveys*, 13(1):71–99, March 1981.

[263] Jennifer Rowley. The wisdom hierarchy: representations of the dikw hierarchy. *Journal of Information Science*, 33(2):163–180, 2007.

[264] Paul Rozin and Adam B. Cohen. High frequency of facial expressions corresponding to confusion, concentration, and worry in an analysis of naturally occurring facial expressions of americans. *Emotion*, 3(1):68–75, 2003.

[265] Per Runeson, Martin Host, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.

[266] James A. Russell and José Miguel Fernández-Dols. *The Psychology of Facial Expression*. Cambridge University Press, 2002.

[267] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering.*, pages 281–292, November 2003.

[268] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2 edition, 2009.

[269] Joni Salminen, Haewoon Kwak, João M. Santos, Soon-Gyo Jung, Jisun An, and Bernard J. Jansen. Persona perception scale: Developing and validating an instrument for human-like representations of data. In *Extended Abstracts of the Conference on Human Factors in Computing Systems*, CHI EA, pages LBW075:1–LBW075:6. ACM, 2018.

[270] Elmar Sauerwein, Franz Bailom, Kurt Matzler, and Hans H. Hinterhuber. The Kano model: How to delight your customers. In Robert W. Grubbström, editor, *9th International Working Seminar on Production Economics*, volume 1 of *WSPE*, pages 313–327, Innsbruck, 1996. Elsevier.

[271] Florian Schaule. Classification of cognitive load using wearable sensors to manage interruptions. Master's thesis, Technical University of Munich, April 2017.

[272] Florian Schaule, Jan O. Johanssen, Bernd Bruegge, and Vivian Loftness. Employing consumer wearables to detect office workers' cognitive load for interruption management. *PACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(1):32:1–32:20, March 2018.

[273] Gerald Schermann, Jürgen Cito, and Philipp Leitner. Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31, March 2018.

[274] Simon A. Scherr, Frank Elberzhager, and Konstantin Holl. Acceptance testing of mobile applications - automated emotion tracking for large user groups. In *5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft, pages 247–251. IEEE/ACM, May 2018.

[275] Harold Schlosberg. The description of facial expressions in terms of two dimensions. *Journal of Experimental Psychology*, 44(4):229 – 237, 1952.

[276] Harold Schlosberg. Three dimensions of emotion. *Psychological Review*, 61(2):81 – 88, 1954.

[277] Kurt Schneider. *Experience and Knowledge Management in Software Engineering*. Springer-Verlag, 2009.

[278] Kurt Schneider. Focusing spontaneous feedback to support system evolution. In *19th International Requirements Engineering Conference*, RE, pages 165–174. IEEE, August 2011.

[279] Jean Scholtz. Adaptation of traditional usability testing methods for remote testing. In *34th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2001.

[280] Donald A. Schön. *The Reflective Practitioner: How Professionals Think In Action*. Basic Books, 1983.

[281] Christian Schüller. *Automated feedback generation in the user-centered software development using the example of mobile applications*. Dissertation, Technical University of Munich, Munich, 2009.

[282] Ken Schwaber. Scrum development process. In Jeff Sutherland, Cory Casanave, Joaquin Miller, Philip Patel, and Glenn Hollowell, editors, *Business Object Design and Implementation*, pages 117–134, London, 1997. Springer London.

[283] Ahmed Seffah, Michel C. Desmarais, and Eduard Metzker. *HCI, Usability and Software Engineering Integration: Present and Future*, pages 37–57. Springer Netherlands, Dordrecht, 2005.

[284] Ahmed Seffah and Eduard Metzker. The obstacles and myths of usability and software engineering. *Communications of the ACM*, 47(12):71–76, 2004.

[285] Marcus Seiler and Barbara Paech. Using tags to support feature management across issue tracking systems and version control systems. In *23rd International Working Conference on Requirements Engineering: Foundation for Software Quality*, volume LNCS 10153, pages 174–180. Springer, 2017.

[286] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017.

[287] Mojtaba Shahin, Peng Liang, and Mohammad Reza Khayyambashi. Improving understandability of architecture design through visualization of architectural design decision. In *ICSE Workshop on Sharing and Reusing Architectural Knowledge*, SHARK, pages 88–95, 2010.

[288] Helen Sharp, Anthony Finkelstein, and Galal Galal. Stakeholder identification in the requirements engineering process. In *10th International Workshop on Database and Expert Systems Applications*, DEXA, pages 387–391, 1999.

[289] Mary Shaw, Jim Herbsleb, Ipek Ozkaya, and Dave Root. Deciding what to design: Closing a gap in software engineering education. In *International Conference on Software Engineering*, ICSE, pages 607–608, 2005.

[290] Tak-Wai Shen, Hong Fu, Junkai Chen, W. K. Yu, C. Y. Lau, W. L. Lo, and Zheru Chi. Facial expression recognition using depth map estimation of light field camera. In *International Conference on Signal Processing, Communications and Computing*, ICSPCC, pages 1–4, August 2016.

[291] Thomas B. Sheridan and Gunnar Johannsen, editors. *Monitoring Behavior and Supervisory Control*. Perseus Publishing, 1976.

[292] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., 1st edition, 1997.

[293] Melanie Stade, Farnaz Fotrousi, Norbert Seyff, and Oliver Albrecht. Feedback Gathering from an Industrial Point of View. In Ana Moreira and João Araújo, editors, *25th International Requirements Engineering Conference*, RE, pages 63–71, Lisbon, Portugal, 2017. IEEE.

[294] Melanie Stade, Marc Oriol, Oscar Cabrera, Farnaz Fotrousi, Ronnie Schaniel, Norbert Seyff, and Oleg Schmidt. Providing a user forum is not enough: First experiences of a software company with crowdre. In *25h International Requirements Engineering Conference Workshops*, REW, pages 164–169. IEEE, September 2017.

[295] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *12th International Conference on Software Engineering*, IASTED, pages 736–743, Innsbruck, Austria, 2013.

[296] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48 – 59, 2014.

[297] Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software*, 127:150–167, 2017.

[298] Klaas-Jan Stol and Brian Fitzgerald. The ABC of Software Engineering Research. *ACM Transactions on Software Engineering and Methodology*, 27(3):1–51, 2018.

[299] Pin S. Teh, Ning Zhang, Andrew D. J. Teoh, and Ke Chen. Recognizing your touch: Towards strengthening mobile device authentication via touch dynamics integration. In *13th International Conference on Advances in Mobile Computing and Multimedia*. ACM, 108–116.

[300] Walter F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.

[301] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9 – 18, 1995.

[302] Marco Torchiano and Filippo Ricca. Six reasons for rejecting an industrial survey paper. In *1st International Workshop on Conducting Empirical Studies in Industry*, CESI, pages 21–26, San Francisco, CA, USA, May 2013.

[303] Zia Uddin. Facial expression recognition using depth information and spatiotemporal features. In *18th International Conference on Advanced Communication Technology*, ICACT, pages 726–731, January 2016.

[304] Lorijn van Rooijen, Frederik Simon Bäumer, Marie Christin Platenius, Michaela Geierhos, Heiko Hamann, and Gregor Engels. From user demand to software service: Using machine learning to automate the requirements specification process. In *25th International Requirements Engineering Conference Workshops*, REW, pages 379–385. IEEE, September 2017.

[305] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999.

[306] Maarten W. van Someren, Yvonne F. Barnard, and Jacobijn A. C. Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press, London, 1994.

[307] Viswanath Venkatesh, Michael G. Morris, Gordon B. Davis, and Fred D. Davis. User acceptance of information technology: Toward a unified view. *MIS Quarterly*, 27(3):425–478, September 2003.

[308] Stefan Wagner, Andreas Goeb, Lars Heinemann, Michael Kläs, Constanza Lampasona, Klaus Lochmann, Alois Mayr, Reinhold Plösch, Andreas Seidl, Jonathan Streit, and Adam Trendowicz. Operationalised product quality models and assessment: The Quamoco approach. *Information and Software Technology*, 62(1):101–123, 2015.

[309] Pascal Welke, Ionut Andone, Konrad Blaszkiewicz, and Alexander Markowetz. Differentiating smartphone users by app usage. In *International Joint Conference on Pervasive and Ubiquitous Computing*, pages 519–523. ACM, 2016.

[310] Roel J. Wieringa. *Design science methodology for information systems and software engineering*. Springer-Verlag Berlin Heidelberg, 2014.

[311] Roel J. Wieringa and Ayşe Moralı. Technical action research as a validation method in information systems design science. In Ken Peffers, Marcus Rothenberger, and Bill Kuechler, editors, *Design Science Research in Information Systems. Advances in Theory and Practice*, pages 220–238. Springer Berlin Heidelberg, 2012.

[312] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Boston, 3rd edition edition, 2011.

[313] Claes Wohlin and Björn Regnell. Achieving industrial relevance in software engineering education. In *Conference on Software Engineering Education and Training*, pages 16–25. IEEE, 1999.

[314] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[315] Wilhelm M. Wundt. *System der Philosophie*. W. Engelmann, 2nd edition, 1897.

[316] Han Xu, Stephan Krusche, and Bernd Bruegge. Using software theater for the demonstration of innovative ubiquitous applications. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 894–897, 2015.

[317] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Conference on Internet Measurement Conference*, SIGCOMM, pages 329–344, 2011.

[318] Bieke Zaman and Tara Shrimpton-Smith. The facereader: Measuring instant fun of use. In *4th Nordic Conference on Human-computer Interaction: Changing Roles*, NordiCHI, pages 457–460. ACM, 2006.

[319] Nityananda Zbil. Discussing usage knowledge through informal communication channels. Bachelor's thesis, Technical University of Munich, March 2019.

[320] Marvin V. Zelkowitz and Dolores W. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, May 1998.

[321] Zhihong Zeng, Maja Pantic, and Thomas S. Huang. *Emotion Recognition based on Multimodal Information*, pages 241–266. Springer, 2009.

[322] Xiang Zhang, Hans-Frederick Brown, and Anil Shankar. Data-driven personas: Constructing archetypal users with clickstreams and user telemetry. In *Conference on Human Factors in Computing Systems*, CHI, pages 5350–5359. ACM, 2016.

[323] Junji Zhi and Günther Ruhe. Devis: A tool for visualizing software document evolution. In *1st Working Conference on Software Visualization*, VISSOFT, pages 1–4, 2013.

[324] Nedaa Zirjawi, Zijad Kurtanovic, and Walid Maalej. A survey about user requirements for biometric authentication on smartphones. In *2nd Workshop on Evolving Security and Privacy Requirements Engineering*, ESPRE, pages 1–6. IEEE, August 2015.