The MALTASE Framework For Usage-Aware Software Evolution

Tobias Röhm

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Forschungs- und Lehreinheit I

Angewandte Softwaretechnik

The MALTASE Framework For Usage-Aware Software Evolution

Tobias Markus Röhm

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:		Univ.-Prof. Dr. Johann Schlichter
Prüfer der Dissertation:	1.	Univ.-Prof. Bernd Brügge, Ph.D.
	2.	Univ.-Prof. Dr. Barbara Paech, Ruprecht-Karls-Universität Heidelberg

Die Dissertation wurde am 23.02.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 01.06.2015 angenommen.

Abstract

Software developers need to know how their software is used to improve it and adjust it to user needs. Helpful usage information includes failure reproduction steps, user skill levels, and deviations of actual usage from expected usage. However, this information is rarely available to developers because of communication gaps between users and developers. Therefore, developers are often unaware of failures and problems users are facing, make potentially wrong assumptions about software usage, and cannot reflect the user perspective in their decisions.

This dissertation investigates the acquisition of usage information by monitoring and analyzing user interactions. It describes the MALTASE framework which monitors high-level user interactions and analyzes them to acquire usage information which is helpful for developers during software evolution tasks.

In a problem case study, we found that developers need the following usage information during software evolution: use cases and user behavior, user goals and user needs, failure reproduction steps, and application domain concepts. We demonstrate the applicability of MALTASE by implementing three ways of exploiting monitored, high-level user interactions: provision of failure reproduction steps, classification of users according to their skills, and detection of deviations between user behavior and use case steps. To investigate the impact of MALTASE, we conducted a controlled experiment and an evaluation case study. The controlled experiment compared failure reproduction with interaction traces to failure reproduction with textual bug reports. It found that developers can reproduce failures based on interaction traces and that inexperienced developers are enabled to reproduce failures they cannot reproduce with textual bug reports alone. The evaluation case study learned skill classifiers from interaction traces of participants with differing skill levels. It found that skill classifiers are able to reliably discriminate between novice and expert users of a specific application (86–93% accuracy) but not between domain skill levels (64–71% accuracy). In a simulation and a user study, we found that MALTASE introduces an overhead of 5 % execution time and that users tolerate this overhead. Overall, our evaluation showed that MALTASE narrows communication gaps between users and developers by automatically acquiring usage information.

Kurzfassung

Um eine Softwareanwendung zu verbessern und an Nutzerbedürfnisse anzupassen, benötigen Softwareentwickler Informationen über Softwarenutzung. Hilfreich sind dabei unter anderem Informationen über Fehlerreproduktionsschritte, Nutzerfähigkeiten, und Abweichungen zwischen tatsächlichem und erwartetem Nutzerverhalten. Allerdings verfügen Entwickler aufgrund von mangelhafter Kommunikation zwischen Entwicklern und Nutzern selten über diese Informationen. Deshalb sind Nutzerprobleme und Softwarefehler Entwicklern häufig unbekannt, sie treffen möglicherweise falsche Annahmen über Softwarenutzung und können die Nutzerperspektive in ihren Entscheidungen nicht berücksichtigen.

Diese Dissertation untersucht die Gewinnung von Nutzungsinformationen durch Aufzeichnung und Analyse von Nutzerinteraktionen. Sie beschreibt das Maltase Framework, welches Nutzerinteraktionen mit einem hohen Abstraktionsgrad aufzeichnet und analysiert. Dadurch gewinnt es Nutzungsinformationen, die für Entwickler während der Softwareevolution hilfreich sind.

In einer Problemfallstudie haben wir herausgefunden, dass Entwickler während der Softwareevolution folgende Arten von Nutzungsinformationen benötigen: Anwendungsfälle und allgemeines Nutzerverhalten, Ziele der Nutzer sowie deren Bedürfnisse, Fehlerreproduktionsschritte und Konzepte der Anwendungsdomäne. Wir verdeutlichen die Anwendbarkeit von Maltase durch die Implementierung dreier Möglichkeiten zur Verwendung von aufgezeichneten Nutzerinteraktionen: der Gewinnung von Fehlerreproduktionsschritten, der Klassifizierung von Nutzern aufgrund ihrer Fähigkeiten und der Erkennung von Abweichungen zwischen Nutzerinteraktionen und Anwendungsfall-Schritten. Wir haben ein kontrolliertes Experiment und eine Evaluationsfallstudie durchgeführt, um die Auswirkung von Maltase zu untersuchen. Das kontrollierte Experiment hat Fehlerreproduktion basierend auf Interaktionsprotokollen mit Fehlerreproduktion basierend auf textuellen Fehlerberichten verglichen. Es ergab, dass Entwickler mit Hilfe von Interaktionsprotokollen Fehler reproduzieren können und dass unerfahrene Entwickler dadurch Fehler reproduzieren können, die sie mit textuellen Fehlerberichten allein nicht reproduzieren können. In der Evaluationsfallstudie haben wir Klassifikatoren für Nutzerfähigkeiten aus Interaktionsprotokollen von Teilnehmer mit verschiedenen Fähigkeitsstufen gelernt. Die Fallstudie ergab, dass die Klassifikatoren zuverlässig zwischen Anfängern und Experten einer spezifischen Anwendung unterscheiden können (86-93 % Genauigkeit), aber nicht zwischen Fähigkeitsstufen bezüglich der Aufgabendomäne (64-71 % Genauigkeit). Eine Simulation und eine Nutzerstudie ergaben, dass Maltase die Ausführungszeit einer Softwareanwendung um 5 % erhöht und dass dies von Nutzern toleriert wird. Zusammenfassend hat unsere Evaluation gezeigt, dass Maltase durch die automatische Gewinnung von Nutzungsinformationen einen Informationsaustausch zwischen Entwicklern und Nutzern ermöglicht.

Acknowledgements

S. D. G.

Many people influenced this dissertation and encouraged me during my dissertation research. Hence, I want to gratefully thank them and acknowledge their support.

I am very grateful to Bernd Bruegge for creating an environment of opportunities and growth, for supporting me in all my ideas and plans, and lots of feedback and discussions. Furthermore, I want to thank Barbara Paech for accompanying my dissertation research, for encouraging discussions, and for her feedback.

I am very grateful to all members of the Chair for Applied Software Engineering for all discussions, feedback, encouragement, and fun. In particular, I want to thank Walid Maalej for recruiting me, for hands-on research teaching, and for the research visit in Hamburg. Similarly, I want to thank Dennis Pagano for all encouragement, for wise and calm answers to my impatient questions, and for all the fun. Furthermore, I want to thank Barbara Reichart, Damir Ismailović, Emitzá Guzmán, Florian Schneider, Helma Schneider, Helmut Naughton, Hoda Naguib, Julian Sußmann, Martin Glas, Michaela Gluchow, Monika Markl, Nitesh Narayan, Uta Weber, Yang Li, and the chair administrators.

I am grateful to all study participants for their time, patience, and feedback as well as to Barbara Reichart, Dennis Pagano, and Rebecca Tiarks for proof-reading parts of this dissertation. Furthermore, I want to thank all my students, collaborators, and co-authors. In particular, collaborators in the research projects FastFix, Punga, ReproFit and URES: Alessandra Bagnato, Alexandra Lungu, Amel Mahmuzic, Barbara Paech, Benoit Gaudin, Bernd Bruegge, Christophe Joubert, Daniel de los Reyes, Dennis Pagano, Emitzá Guzmán, Javier Cano, João Coelho Garcia, Miguel Juan, Miriam Schmidberger, Patrick Bürgin, Rainer Koschke, Rebecca Tiarks, Sergio Szamarripa, Stefan Nosović, Tom-Michael Hesse, Walid Maalej, and Zardosht Hodaie. Similarly, the students Iulia Gaina, Martin Stoll, Nadeem Ahmed, Nigar Gurbanova, and Stefan Theiner.

In addition, I want to thank Alexander Lehmann, Sebastian Vogl, and Sergej Trushin for lunch and encouragement. Last but not least I want to thank my family - Hans, Rose, Stefan, and Johannes - and my girlfriend Kerstin for their support and status enquiries.

Contents

Chapter 1

Introduction

> "Pay attention to what users do, not what they say. Self-reported claims are unreliable, as are user speculations about future behavior."
> Nielsen [128]

Interactive software applications are developed by software developers for end users. Hence, interactive applications usually have at least two stakeholders, one user and one developer. The notion of software user has changed a lot in the last decades: while batch processing systems had no users or only a few expert users, smartphone apps are used today by almost everybody. Software users have become a heterogenous group of people with differing usage purposes, differing skills, and differing usage contexts. For example, Microsoft Excel is used for spreadsheet calculations, role-playing games[1], or drawing Japanese art[2].

Users have not only become more heterogenous but also more powerful stakeholders. Users can harm the reputation of a software application using modern communication channels such as social media, forums, or app store feedback. For example, web pages like ihatelotusnotes.com[3] or dreckstool.de[4] collect negative user feedback. Also, users can unite and campaign against a software vendor using modern communication channels. For example, editors and film makers launched a successful campaign for reinstating Final Cut Studio 3 and continue support of the older version after Apple released a new version with paradigm-shifting changes[5]. Finally, users as consumers have a high market power in competitive software markets. For example, modern app stores offer users a variety of apps. In case of several similar apps, users buy the app which is most appealing to them.

This change of software users is reflected in different perceptions of users and user involvement among software developers and researchers. Two contrasting quotes highlight this aspect: Dijkstra states that "the notion of 'user' cannot be precisely defined, and therefore has no place in computer science or software engineering" [39], indicating that users should be ignored by developers. In contrast, Begel and Zimmermann [15] found in an empirical study that the question "How do users typically use my application?" is the most important question of developers, indicating that

[1] http://gizmodo.com/5992533/awesome-accountant-made-an-rpg-inside-microsoft-excel (Accessed Feb 2015)
[2] http://kotaku.com/old-japanese-man-creates-amazing-art-using-excel-wait-499616608 (Accessed Feb 2015)
[3] http://www.ihatelotusnotes.com/ (Accessed Feb 2015)
[4] http://www.dreckstool.de (Accessed Feb 2015)
[5] http://www.petitiononline.com/finalcut/petition.html (Accessed Nov 2014)

information about users and software usage is an important piece of information for developers. The increased power of users calls for emphasizing the user perspective [19] and user involvement in software engineering. User involvement aims at eliciting user needs and continuous exchange of information between developers and users and in order to develop successful software applications. Studies show a positive correlation between user involvement and system success [4, 70].

Several techniques to involve users have been developed in the requirements engineering field: interviews [140], workshops [140], focus groups [140], observations [140], questionnaires [140], and user studies with prototypes [23] are used to elicit information about users and user needs and to test initial application designs; personas [59], scenarios [140], use cases [140], and user stories [32] are used to document the information gained. Furthermore, users or user representatives are actively involved in software development methodologies such as user-centered design [59]. Also, users of deployed applications can provide feedback to developers via e-mails [136], bug reports [203], app store feedback [138], or integrated feedback mechanisms [136]. The big number and heterogeneity of software users poses challenges to existing user involvement methods: Interviews, workshops, focus groups, or observations are difficult to scale to a large number of users because of the required effort. Users in interviews speculate about future needs and future behavior and those speculations might be wrong [128]. The representation of users by a single product owner or few personas might not be representative when the user group is heterogenous. The vast number of potential usage contexts makes it difficult to anticipate and test all usage contexts in which a software application might be used [131]. In addition, many user involvement methods are usually employed during the design or development phase of an application but not during the evolution and operation phase. User feedback mechanisms often require users to proactively share feedback and describe their feedback manually.

An approach which complements existing user involvement methods and addresses these challenges has to fulfill the following requirements: It should consider real users and their behavior to avoid speculations about future behavior and avoid loosing information by representing a heterogenous user group by few representatives. It should be automated to scale to a large number of users and user groups. It should be employed during the evolution and operation phase to supply developers with usage information. And finally it should capture user behavior and context automatically to relieve users from manually describing such information.

This dissertation investigates an approach which complements existing user involvement methods and fulfills these requirements. It follows Nielsen's advice to "pay attention to what users do, not what they say" [128] by monitoring interactions of users with an application. Then it analyzes monitored interactions to extract usage knowledge which is helpful for developers during software evolution. The next section pinpoints the problem addressed in this dissertation.

1.1 Problem Statement

Recent empirical studies identified a strong interest of developers in information about software usage: Begel and Zimmermann [15] found that information about

"how users typically use an application" and "the parts of a software product most used and/ or loved by customers" are the two most important questions of software developers for data analysts. Buse and Zimmermann [26] describe a decision scenario "understanding customers" which "leverages information about customer behavior when making decisions". In an observation study, we found that developers are interested in "they way end users use an application" but rarely have access to this information [154].

Several communication gaps between developers and users hinder the acquisition of usage information by developers. Developers are usually not present when users employ an application. Therefore they have no first-hand information about software usage and have to acquire usage information indirectly. Maalej et al. [115] identify communication gaps between developers and users as reason for missing information of developers about software usage. Abelein and Paech [3] found the following reasons for communication gaps: lack of motivation of developers or users, lack of common language between business and IT, and lack of appreciation between business and IT. Schinzel [159] found that computer science students lose their user-centric perspective during their education and adopt a developer-centric perspective, creating a cultural difference between developers and users. Mann [119] identified nine gaps between developers and users, among them a communication gap, a cultural gap, and a relationship gap. Besides these gaps in the direct communication between developers and users, the current practices of indirect communication, i.e. collection of user feedback, are inefficient [136]. These communication gaps and feedback collection practices lead to the situation that software developers are interested in information about software usage but rarely have such information.

Missing usage information has negative effects for developers as they are unaware of failures and problems users are facing, make potentially wrong assumptions about software usage, and do not reflect the users' perspective in their decisions. For example, failure reproduction and bug fixing are so hard and time-consuming because of missing usage information [117].

Summarizing, the problem statement of this dissertation is the following:

> **Problem Statement**
> Software developers are interested in information about end user software usage but rarely have such information because of communication gaps between developers and users.

1.2 Research Approach

This dissertation addresses the problem of missing usage information by monitoring and analyzing user interactions. Our main hypothesis is the following:

> **Main Hypothesis**
> High-level user interactions provide semantic information which helps software developers to understand how their software is used and to evolve their software according to user needs.
> Monitoring high-level user interactions represents a way to automatically inform software developers about software usage.

"High-level user interactions" denote user interactions on a high level of abstraction like user commands or manipulations of work artifacts. They contrast with low-level user interactions like keyboard or mouse actions. We hypothesize that high-level user interactions carry semantic information and allow developers to understand user behavior. Moreover, we hypothesize that an understanding of software usage helps developers in their evolution tasks. For example, information about reproduction steps enables developers to reproduce and fix bugs.

To study our main hypothesis, an infrastructure for capturing, storing, mining, and presenting high-level user interactions is necessary. For this purpose, we developed the MALTASE framework. MALTASE is a shorthand for "Monitoring, Analysis and ExpLoiTation of User InterActions in Software Evolution". The purpose of the MALTASE framework is to monitor high-level user interactions, automatically analyze monitored interactions, and provide developers with usage information and support them in software evolution tasks.
MALTASE fulfills the requirements described above: it considers the real user behavior and therefore addresses user speculations about future usage of the target application, it is automated and therefore scalable to a large number of users, it can be employed during software evolution, and it automatically captures user behavior and therefore relieves users from manually describing it. Furthermore, MALTASE follows Nielsens' advice to "pay attention to what users do, not what they say. " [128], considers user input as a first-order concern as advocated by Maalej et al. [115], and builds on the general framework for user involvement in software evolution suggested by Maalej and Pagano [116].

To study our main hypothesis, we investigate the following research questions in this dissertation:

> **Research Question RQ1:**
> What are information needs regarding software usage of developers during software evolution?

To investigate RQ 1, we conducted an exploratory case study and analyzed observation protocols and interview minutes of 21 developers during program comprehension tasks.

> **Research Question RQ2:**
> How can high-level user interactions be monitored with acceptable performance overhead?

To investigate RQ 2, we implemented the MALTASE framework and investigated its performance overhead by a simulation and a user study. We simulated user interactions with an instrumented and an uninstrumented target application to measure the performance overhead introduced. Furthermore, we had five users work with an instrumented version of a target application and report their experiences.

> **Research Question RQ3:**
> What is the impact of MALTASE in software evolution?
> More specifically, can MALTASE be employed to acquire failure reproduction steps, user skill levels, and deviations from expected user behavior?

To investigate RQ 3, we describe three applications of the MALTASE framework in software evolution. These framework applications refine our main hypothesis and target one specific type of usage knowledge. First, MALTASE-based failure reproduction presents monitored user interactions preceding software failures to developers during bug fixing. We hypothesize that developers can reproduce failures and fix bugs based on monitored user interactions. Second, MALTASE-based skill detection detects user skill levels based on monitored user interactions. We hypothesize that that information about user skill helps developers to evolve the application and help system according to user needs and enables an intelligent application to adapt to the current user. Third, MALTASE-based use case testing compares monitored user interactions with use case steps. We hypothesize that differences between both help developers to identify software improvements and use case updates.
We evaluated MALTASE-based failure reproduction and MALTASE-based skill detection. More specifically, we evaluated MALTASE-based failure reproduction in a controlled experiment by comparing failure reproduction with monitored user interactions to failure reproduction with textual bug reports. Moreover, we evaluated MALTASE-based skill detection in a case study by learning skill classifiers from monitored user interactions of users with different skill levels.

1.3 Scope

Monitoring, analyzing, and exploiting user interactions in software evolution is a broad topic which can be investigated from many different perspectives. Hence, this section narrows the focus of this dissertation.

This dissertation focuses on a single, interactive software application. We implemented and evaluated MALTASE for desktop applications which exhibit a WIMP GUI. We hypothesize that MALTASE is applicable to other kinds of interactive software, but investigation of this hypothesis is out of scope of this dissertation.

Software engineering consists of different activities and phases. This dissertation focuses on the software evolution phase after the initial deployment of an application. We hypothesize that MALTASE can be employed during software testing and prototype-based software development, too, but investigation of this hypothesis is out of scope of this dissertation.

This dissertation focuses on a software development context where developers and users are different people, excluding contexts such as end user development and developers building software for their own use. Only in this context the communication gap between developers and users exists and developers have missing knowledge about software usage.

As MALTASE monitors user interactions, privacy concerns arise because of the sensitivity of usage data. We argue that it suffices for MALTASE to monitor user interactions anonymously ("user did X") without personal information like user identity ("user Tobias did X"). Further treatment of privacy concerns is out of scope of this dissertation. But future work should study privacy concerns regarding user interaction monitoring.

Researchers and practitioners test the usability of applications or new interaction mechanisms by monitoring and analyzing user interactions. In contrast, this dis-

sertation focuses on the exploitation of user interactions in software evolution to address the problem of missing usage information. We hypothesize that user interactions monitored by MALTASE can be exploited for usability analysis purposes, too, but investigation of this hypothesis is out of scope of this dissertation.

1.4 Structure

Chapter 2 defines terminology and presents relevant background information from the fields of human-computer interaction and software engineering.

Chapter 3 presents an exploratory case study about information needs of developers during software evolution with a focus on usage information. Furthermore, the case study investigates when and why developers put themselves in the role of users by interacting with the user interface [154]. Chapter 3 describes the case study design, summarizes results regarding usage information needs and developer behavior, relates findings to related work, and discusses conclusions and future work.

Chapter 4 describes the MALTASE framework in detail. It presents use cases and requirements and describes a model of user interactions, users, applications, and usage contexts. Furthermore, it presents the architecture of the MALTASE framework consisting of four layers: Monitoring & Information Extraction layer, Data Storage & Transfer layer, Processing & Analysis layer, and Presentation & Integration layer. Then, it discusses the usage of the MALTASE framework in three usage scenarios which are software evolution, software testing, and prototype-based software development. Finally, it describes related approaches and tools to monitor, analyze, and exploit user interactions.

Chapter 5 illustrates the applicability of MALTASE by describing three framework applications, i.e. ways to exploit monitored, high-level user interactions in software evolution: the extraction of reproduction steps for software failures (MALTASE-based failure reproduction), the classification of user skill levels (MALTASE-based user skill classification), and the comparison of monitored user interactions to use case steps (MALTASE-based use case testing). Chapter 5 motivates each framework application, describes its implementation, and discusses related work.

Chapter 6 presents an empirical evaluation of the MALTASE framework. It describes a simulation and a user study to investigate the performance overhead and user acceptance of MALTASE monitoring. To investigate the impact of MALTASE, it describes the evaluation of two framework applications: a controlled experiment evaluating MALTASE-based failure reproduction as well as an evaluation case study evaluating MALTASE-based skill detection. Finally, it discusses important findings and implications.

Chapter 7 concludes the dissertation. It summarizes the contributions of this dissertation, discusses general issues, and sketches future work.

Chapter 2

Foundations

This chapter presents relevant background knowledge for this dissertation. It provides definitions for important terms (Section 2.1) and introduces relevant concepts from the fields of human-computer interaction and software engineering (Section 2.2).

2.1 Terminology

The following terms are used in this dissertation:

Bug "The mechanical or algorithmic cause of an error." [23]

Case Study "An empirical method aimed at investigating contemporary phenomena in their context." [157]. Easterbrook et al. [42] identify two types of case studies: "Exploratory case studies are used as initial investigations of some phenomena to derive new hypotheses and build theories" and "confirmatory case studies are used to test existing theories".

Controlled Experiment "An investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables." [42]

Developer "Developers create a software application." [146] (Slightly adapted). Developers are one type of **stakeholder** for a software application.

Dynamic Analysis "The analysis of data gathered from a running software application." [33] (Slightly adapted).

End User see **User**

Error "State of the software application such that further processing will lead to a failure." [23] (Slightly adapted)

Failure "Deviation of the observed behavior from the specified behavior." [23]

Interactive Application A software application which exhibits a **user interface** and is driven by **user interactions**. There are different types of interactive applications such as desktop **GUI** applications, web applications, and smartphone apps. This type of software is also called "event-driven software" [74]. This dissertation focuses on desktop GUI applications.

Program Comprehension "The activity of understanding how a software application or a part of it works." [117] (Slightly adapted)

Reproduction Steps "The steps necessary to create the failure." [201] (Slightly adapted). While reproduction steps can target different sources of non determinism (Zeller [201]), this dissertation focuses on **user interactions** as **reproduction steps**, i.e. user interactions triggering a **failure**.

Skill "The ability to do something well; expertise"[1]. Nielsen [127] differentiates between three types of skill: computing skill (skill regarding computing in general), (software) application skill (skill regarding a particular software application), and domain skill (skill regarding a particular knowledge domain).

Stakeholder "Stakeholders affect or are affected by the software application." [146] (Slightly adapted)

Software Analytics "Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions." [121]

Software Developer see **Developer**

Software Evolution "The continual change of a software application." [50] (Slightly adapted) or "The application of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version" [28].

Software Maintenance "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." [1]

Software Usage Analytics One type of **software analytics** which analyses **usage information** of a software application. Similar to **dynamic analysis** (see discussion below). This dissertation investigates software usage analytics.

Software User see **User**

Usage Information Runtime data gathered from the **interaction** of real, human **users** with a software application.

Use Case "A general sequence of interactions between one or more actors and the software application." [23] (Slightly adapted)

User "The people who (will) use the delivered software application." [23] (Slightly adapted) Users are one type of **stakeholder** for a software application.

User Interaction "Communication between user and software application" [40] where a **user** provides user input to a software application via the **user interface.**

[1] http://www.oxforddictionaries.com/ (Accessed Feb 2015)

User Interface "A way for humans to interact with a software application"[2]

Graphical User Interface (GUI) "A human-computer interface that uses windows, icons and menus and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well)."[3] (Slightly adapted) We use the term GUI and WIMP ("windows, icons, menus and pointers") interchangeably. GUIs are one type of **user interface.**

User Involvement "A systematic exchange of information between (prospective) users and developers with the common goal to maximize application usefulness in a specific context." [135] (Slightly adapted)

2.2 Background

This section discusses relevant concepts and theories from human-computer interaction and software engineering.

2.2.1 Differences and Gaps Between Developers and Users

Differences and gaps between software developers and software users have been identified and described by several researchers.

Different Conceptual Models of Developers and Users

Figure 2.1 depicts conceptual models of users and developers as described by Norman [129]. Each developer forms a conceptual model of an existing or future software application, i.e. the developer's conception of the look, feel, and operation of the application. Afterwards, the developer implements the application according to his or her conceptual model. Because users usually cannot communicate to developers while using an application, they have to build an own conceptual model of the application. To accomplish this, the user interacts with the application and analyzes the application image, i.e. user manuals, tutorials, online information, and other forms of documentation. As developers and users usually cannot communicate directly, the whole communication is burdened on the application image. According to Norman, a developer implicitly assumes that the user's conceptual model is identical to the developer's conceptual model.
This situation illustrates the importance of developer-user communication and user feedback: It allows the developer to test whether the user's conceptual model is identical to the developers', to test whether developer assumptions are correct, and to ensure that the application is understandable and usable. The user's conceptual model may change over time because of changes in usage context or user needs. While hardware products cannot react to these changes, software can and has to adapt to them [23]. Hence, continuous developer-user communication and continuous user feedback are important during software evolution to detect changes in usage context and user needs.

[2]Linux Information Project, http://www.linfo.org/gui.html (Accessed Feb 2015)
[3]Linux Information Project, http://www.linfo.org/gui.html (Accessed Feb 2015)

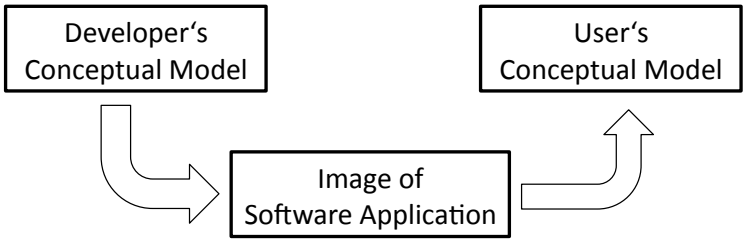


Figure 2.1: Conceptual Models of Developers and Users (Adapted from [129])

Different Domains of Developers and Users

Developers and users are usually knowledgeable in different domains: While users are familiar with the application domain of a software application, developers are familiar with the solution domain. The application domain "represents all aspects of the user's problem, including the physical environment, the users, and their work processes" [23], while the solution domain is "the space of all possible applications" [23] and addresses design and implementation details. Usually, users do not have knowledge about the solution domain and developers have limited knowledge about the application domain. Baster et al. ([11] as cited by [188]) call this mismatch the "business-technology gap": "technology specialists lack domain expertise while business users lack technology skills". As described by Tuffley [187], the fact that developers and users are knowledgeable in different domains leads to "a fundamental difference in mindset": While developers "naturally have a tendency towards a technical perspective" and their "view of the world derives from a technological perspective", users "typically will have a non-technical view of the world" [187]. Abelein and Paech [3] describe the "lack of common language between Business and IT" as a factor for communication gaps between users and developers. Tesch et al. [183] found that shared knowledge between developers and users - knowledge of users about development and knowledge of developers about the application domain - has a significant impact on the success of information system development projects.
Because of knowledge in different domains, users cannot give feedback or discuss solution domain phenomena because they are usually not knowledgeable about them. Furthermore, developers should be knowledgeable in the application domain to understand user needs and usage context. Therefore, developers have to communicate wit users to elicit application domain knowledge.

Cultural Differences Between Developers and Users

Developers and users are not only knowledgeable in different domains, the culture of both groups is also different. Grudin [66] dicusses that physical distance as well as barriers of class, culture, or language separates both groups. Grindley ([65] as cited by [188]) explains that "the culture gap is manifested by differing approaches to motivation, goals, language, and problem-solving". Similarly, Tuffley [187] describes that "software developers have their own culture with their own priorities, preoccupations, and ways of doing things". Taylor-Cummings and Feeny ([181] as cited by [188]) define the cultural gap as "diverse interests, conflict, and power". Fur-

thermore, they recognize and discuss "organizational culture", i.e. the sub-culture within a broader organizational culture and its influence on the cultural gap. Mann ([119] as cited by [188]) observed that "IT personnel have different personality traits than the general population". Schinzel [159] found that computer science students gradually adopt a developer perspective: While they look at software from a user perspective at the beginning of their studies, their perspective changes to a developer perspective during the course of their studies. Abelein and Paech [3] found that "lack of appreciation between Business and IT" is a factor for communication gaps between users and developers.
The cultural gap between developers and users has serious consequences: In two surveys, half of the participating IS directors mentioned the cultural gap as their biggest challenge ([65, 181] as cited by [188]). Taylor-Cummings and Feeny ([181] as cited by [188]) blame the cultural gap as one reason for failing IT projects.

Gaps Because of Organizational Structure or Development Process

The organization of a software development entity and the development process can create additional gaps between developers and users. Grudin [66] identifies three different categories of software development as contexts for user involvement: contract development, product development, and in-house/ customized development. Because users are identified late during product development, developers cannot communicate with them in the early development phase. Furthermore, Grudin [66] discusses that the use of specification documents imposes a wall between developers and users and that software companies might shield developers from users. Bjarnason et al. [18] found that development work is distributed among many people and roles which vary over the lifetime of a development project. Hence, it is hard to achieve common knowledge among developers in different roles, especially at handover points when work is passed to another role. Abelein and Paech [2] describe a communication gap in large IT projects: Because of high project and software complexity as well as traditional project management and software engineering methods, development cycles are very long. Those long development cycles induce long waiting periods for business users, i.e. time periods where they do not get feedback about the status of the IT project.

Classification of Developer-User Gaps

According to Mann [119], gaps between developers and users can be categorized in nine ways:

- Perspective Gap: "When the viewpoint of one group is incomplete. For example, IT sometimes forgets that systems must provide value to the business or users sometimes forget that IT is only a tool and not a cure-all." [119]

- Ownership Gap:
  "Often IT feels ownership over the infrastructure and users feel ownership over business processes. This can lead to territorial conflicts." [119]

- Cultural Gap: "When the different groups have different traits, values, working behaviors, and/ or priorities because each group attracts certain kinds of individuals." [119]
- Foresight Gap: "When one group is better able to see the future but cannot convince the other group. For example, IT may foresee that a user solution will not work from a technical standpoint or users may determine that a system will not be accepted." [119]
- Communication Gap: "When one group fails to understand what the other means. For example, users might feel that IT speaks jargon or IT fails to translate user needs into useful systems because they don't fully understand business processes." [119]
- Expectation Gap: "When end-users have unrealistic expectations of what IT can do or IT promises more that it can deliver." [119]
- Credibility Gap: "When track record of IT side is poor, e.g. because of failed previous development projects." [119]
- Appreciation Gap: "When one group feels the other group does not recognize their value. For example, IT may feel that their hard work and contributions go unnoticed." [119]
- Relationship Gap: "When the two groups do not interact frequently and effectively enough. Each group's pre-judgements of the other group never become resolved and the relationship becomes 'us' versus 'them'." [119]

These findings about differences and gaps between developers and users illustrate a dilemma: On the one hand, user involvement and developer-user communication is necessary to bridge different conceptual models, to learn about the application domain, and to ensure that a software application meets user needs. It has been shown that good developer-user communication is a critical success factor for software development projects [82] and that user involvement during software development has a positive effect on system success [4]. On the other hand, gaps between developers and users complicate this communication process, often leading to poor communication and therefore failed IT projects or bad software quality [187].
During software evolution, the developer-user gap is worsened by the fact that developers are not present during software usage [129] and therefore have only limited and indirect information about it. But software applications must evolve with changing user needs to remain useful [23] which requires information about software usage and changing user needs.

2.2.2 User Involvement in Software Evolution

User involvement, the "systematic exchange of information between (prospective) users and developers" [135], has been studied during software development [136] with a focus on early or late development phases [2]. Techniques have been designed

to elicit user needs and requirements and to test whether software applications fulfills them. Furthermore, users have been actively involved in software development processes by approaches such as participatory design or user-centered design. It has been shown that user involvement has a positive effect on software success, but the relationship between both is not always positive and depends on many factors [4, 10]. But little is known how users can be involved during software evolution and how developers can leverage post-deployment user feedback [136].

Empirical Studies About User Involvement in Software Evolution

Pagano and Bruegge [136] report on a case study about user involvement in small and middle-sized companies. They found that "users and developers are disconnected due to communication gaps in user feedback channels", "users are not systematically involved during software evolution", and that "apart from error reports there is no commonly agreed practice for user feedback in software evolution". Furthermore, they found that "developers need real-world data from user environments to complement tests and to align development efforts with feature importance" and that "user feedback [...] serves as real-world usage data". While highlighting the importance of real-world usage data for developers, the use of feedback submitted by users reveals a limitation: Because of the problems with user feedback such as poor quality and missing context, the information reaching developers remains incomplete and might be influenced by the reporting user. This problem can be addressed by collecting real-world usage data automatically which is the idea of MALTASE.
Heiskari and Lehtola [73] report on a case study about user involvement in a software company. They report that "customers are often seen as the most important stakeholders as they are paying for the system" and that "the biggest challenges are that there is too little user information available, the information is not accessible nor utilized efficiently, and there is not enough interaction with the end users in general". Additionally, they observed that different departments collect different types of user feedback which requires knowledge exchange between departments: "Support provides user feedback from already shipped products and customer involvement provides feedback from products in their beta phase".
Ko et al. [98] report on a case study of a development team handling post-deployment user feedback which requires changes of the application. They identified two dimensions of change requests: First, the scope of developer assumptions behind code, i.e. "how much code would have to be changed to modify an assumption". Changing a local assumption requires modification of a local part of the code base while changing a global assumption requires modification of a significant part. Second, the expected user support of a change request, i.e. "the extent to which a user expectation (change request) was believed to be shared". A particular change request can be supported by the majority of all users or only by a minority of users. When a particular change request is only backed by a minority, it might be that another minority group has a different change request and that the change requests of both groups conflict. Ko et al. [98] found that all types of changes besides changes regarding a local assumption and backed by the majority of users are difficult to address for a development team.

User Feedback in Software Evolution

Collecting and analyzing user feedback has been studied by several researchers. Several researchers, e.g. [7, 115, 116, 134, 135, 200], propose to involve users and collect user feedback by integrating a user feedback mechanism into a software application, i.e. a mechanism which allows users to give feedback within an application. The integration into the software application makes it easier for users to provide feedback and allows to automatically capture the context of user feedback [115].

Similarly, user feedback can be collected by feedback collection systems which are independent and not integrated into a particular application. We distinguish between the bug and issue reporting infrastructure of an application on the one side and dedicated feedback systems on the other side. If the bug and issue reporting infrastructure of an application is accessible for users, they can provide feedback by creating tickets. In this situation, the issue and bug reporting infrastructure usually serves the dual purpose of collecting user feedback and organizing the work of developers or support personnel. Zimmermann et al. [203] studied such bug reporting practices in open source software development.
Dedicated, application-independent feedback collection systems using mobile devices have been proposed [142, 161, 162, 163, 198]. Users can give feedback by entering text or taking pictures using a feedback app. Then, this user feedback is sent to developers. Such an approach is not limited to collect feedback about an existing version of the application and users can also report information about the usage context or requirements in general. Hence, such approaches enable continuous requirements engineering driven by users [162].

Aforementioned techniques collect feedback directly from individual users. Because of the widespread use of app stores, social media, and online communities, users express their opinion about and experiences with an application in app store reviews, blog posts, tweets, or forum posts. Several researchers proposed mining approaches to analyze such online resources and extract user feedback and requirements from them [57, 68, 86, 87, 130, 138].

As Maltase captures user interactions, it is related to the feedback collection approaches which integrate into a software application and collect interactions as context information for user feedback [115, 116, 135]. Maltase is complementary to feedback collection mechanisms as it collects user interactions but not user feedback.

2.2.3 User Interfaces of Software Applications

User interfaces are the face of a software application. Users interact with the user interface to provide input, to analyze the application's reaction on their input, and to control the application. The user interface is often the only part of a software application which users see. Hence, users form their conceptual model as well as their impression of a software application based on its user interface. Many different types of user interfaces exist. Rogers et al. [155] distinguish between 20 different types of user interfaces: command-based, WIMP/ GUI, multimedia, virtual reality, information visualization, web, consumer electronics and appliances, mobile, speech, pen, touch, air-based gesture, haptic, multimodal, shareable, tangible, augmented

and mixed reality, wearable, robotic, and brain-computer. These categories describe properties of a particular user interface and are not mutually exclusive. For example, smart phones have a user interface which can be categorized as mobile and gesture.

The WIMP GUI

This dissertation focuses on target application with a WIMP (Windows, Icons, Menus, Pointing device) GUI (graphical user interface). The WIMP GUI was introduced with the Xerox Star computer ([171] as cited by [155]) and evolved to the standard user interface for desktop applications. Rogers et al. [155] define the following elements of a WIMP GUI: "Windows (that could be scrolled, stretched, overlapped, opened, closed, and moved around the screen using the mouse), Icons (to represent applications, objects, commands, and tools that were opened or activated when clicking on), Menus (offering lists of options that could be scrolled through and selected in the way a menu is used in a restaurant), Pointing device (a mouse controlling the cursor as a point of entry to the windows, menus, and icons on the screen), Docks (a row or column of available applications and icons of other objects such as open files), and Rollovers (where text labels appear next to an icon or part of the screen as the mouse is rolled over it)."

Figure 2.2 shows a screenshot of the CASE tool MOSKitt as an example of a WIMP GUI. MOSKitt was used as target application in the evaluation of this dissertation. MOSKitt exhibits a WIMP GUI which consists of a main window with several sub-windows, a menu bar, a tool bar, a dock representing open files (below the tool bar), context menus (not visible), and rollovers (not visible). Additionally, MOSKitt features a palette to choose diagram elements (right) and a canvas to place and manipulate diagram elements via drag and drop (center). MOSkitt is mainly operated using a mouse. For example, users can select diagram elements from the palette, add selected diagram elements to the diagram canvas, or manipulate properties of diagram elements.

2.2.4 Nature of User Interactions

This section discusses important aspects of user interactions which are monitored and analyzed by the Maltase framework. Figure 2.3 illustrates a spectrum of events according to their duration as described by Hilbert and Redmiles [79]. The horizontal axis shows the event duration in seconds on a logarithmic scale. The duration of an UI event is between 10 milliseconds and 1 sec. Event types with a short duration such as UI events usually occur with a higher frequency and are referred to as "high frequency band events". Likewise, event types with a long duration such as project events usually occur on a lower frequency and are referred to as "low frequency band events". Moreover, events of a particular frequency band are often composed of several events from a higher frequency band. This spectrum situates UI events in an overall spectrum of events.

User interactions can be considered at different levels of abstraction. Figure 2.4 provides an overview of different abstraction levels as described by Hilbert and Redmiles [79]. Physical interactions of users with input devices form the lowest abstraction level. The next abstraction level are the interrupts produced by input

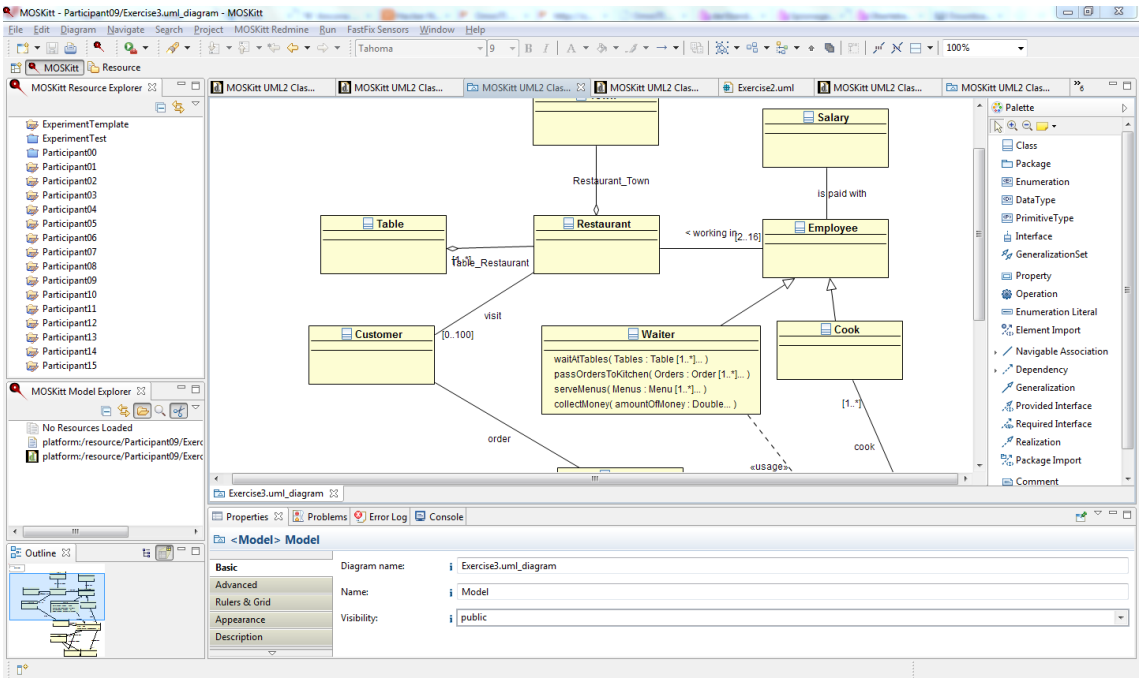


Figure 2.2: MOSKitt User Interface

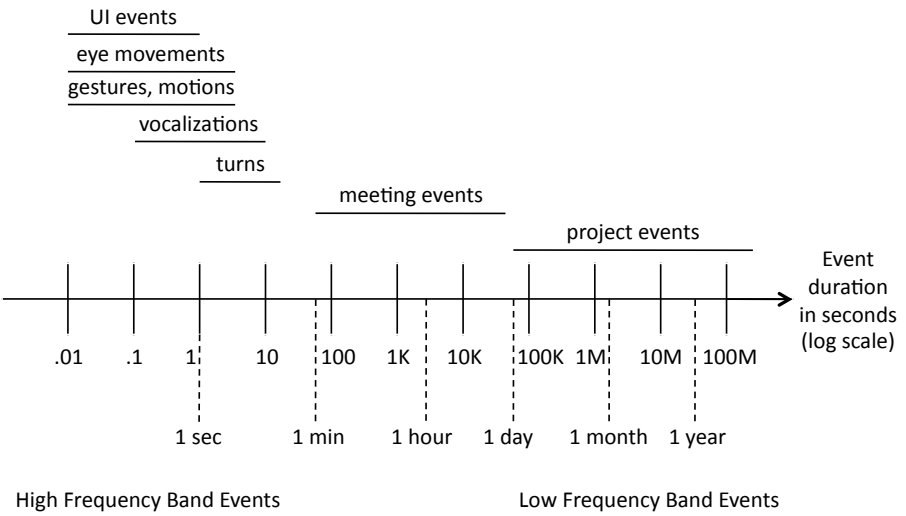


Figure 2.3: Spectrum of HCI Events (Source: [79])

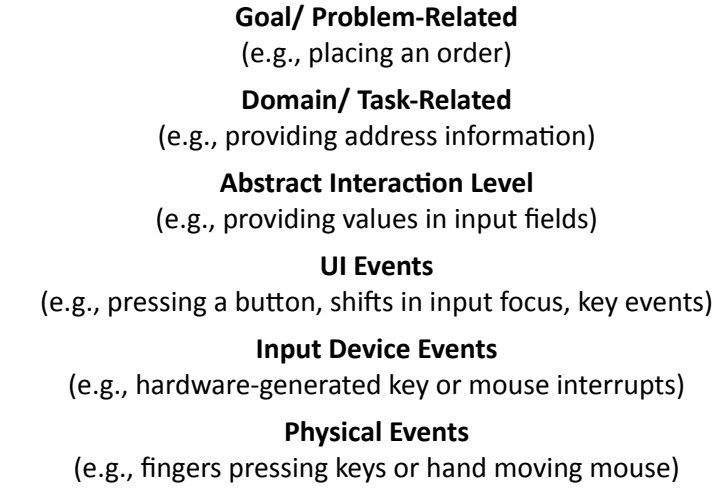


Figure 2.4: Abstraction Levels of User Interactions (Source: [79])

devices as reaction to physical events. On the next abstraction level, UI events, input device events are associated with windows or other elements of the GUI. Examples of UI events are button presses, menu selections, focus events in input fields, and window activations. The next level are abstract interaction events which are not generated by the user interface and which usually consist of several UI events. For example, entering text in a textfield consists of several key presses on the keyboard. Finally, domain/ task-related and goal/ problem-related events indicate steps in a user task or progress towards a user goal.

2.2.5 Processing of User Interactions

This section describes how user interactions are processed by computer systems. It sketches a processing pipeline which consists of all hardware and software components involved in processing a single user interaction such as a mouse click. Afterwards, it describes how user interactions are handled in an event-driven GUI application using Eclipse SWT as example. This section provides background information for the MALTASE framework which monitors user interactions and whose sensors target SWT applications.

Processing Pipeline of User Interactions

We follow Tanenbaum [176] when discussing hardware and operating system components involved in processing user interactions. This description is rather abstract and implementation details might vary from system. Figure 2.5 shows a logic view of the hardware and software components involved in processing a single user interaction such as a mouse click. Each hardware device consists of a mechanical part and an electronic part. The mechanical part is the device itself, e.g. a computer mouse with buttons. The electronic part is the device controller. It consists of registers for command and status information and memory blocks for data transfer to and from the system. On the system side, there are four layers of software components in-

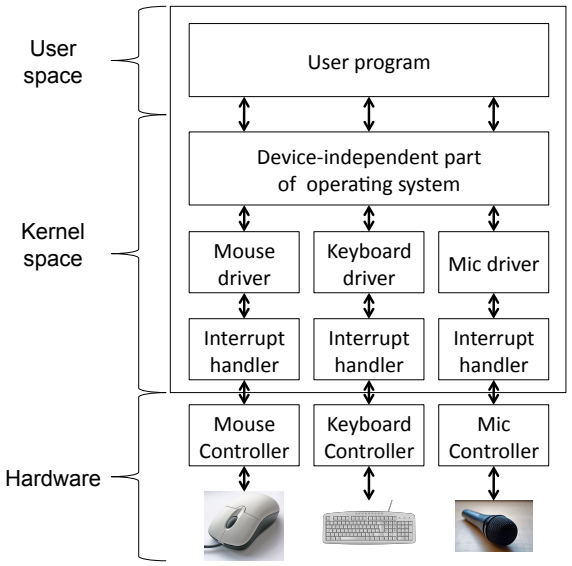


Figure 2.5: Processing Pipeline of User Interactions (Logic view)
Adapted from Tanenbaum [176]
Device controllers and device drivers communicate via system buses

volved in processing user interactions using interrupt-driven I/O: interrupt handlers, device drivers, the device-independent part of the operating system, and the user program. Interrupt handlers manage the interrupts which control the communication between device controller and device driver. Device drivers are device-specific code controlling a particular input device. They initialize the device at startup and control read and write operations from and to the device. The device-independent part of the operating system implements device-independent functionality such as error handling or buffering. Finally, the device-independent part determines the target application of an interaction, generates GUI events representing the interaction, and places those events in the event queue of the target application.

For example, a hardware sensor detects when a user clicks on the left button of his or her mouse. The mouse device controller triggers issues an interrupt to inform the system about the user interaction and sends a message with detail information. In this example, the message consists of the three parameters $\triangle$x, $\triangle$x, and buttons. $\triangle$x and $\triangle$y denote the relative movement of the mouse since the last report and buttons the status of the mouse buttons. The responsible interrupt handler is determined via the interrupt vector table and executed. It handles the interrupt by pausing the active process, saving status information of the active process, reading the message from the mouse controller, and unblocking the mouse device driver. The mouse device driver processes the message from the device controller and passes it to the device-independent part of the operating system. This component determines the target application of the mouse click, creates a GUI event representing the click, and places this GUI event in the event queue of the target application. Other types of user input devices use similar mechanisms depending on supported interactions.

Processing of User Interactions in SWT Applications

GUI applications are driven by interactions of users with the user interface. Each user interaction is represented as an event which is processed by the application. Hence, GUI software is also called event-driven software. In the following, we discuss how GUI applications handle events. We use the Standard Widget Toolkit (SWT) as example because applications using toolkit are studied during the evaluation of this dissertation. Other GUI toolkits use similar mechanisms. The following discussion is based on the Eclipse Programmers Guide[4], the Eclipse SWT javadoc[5], and the book by Vogel [194].

The Standard Widget Toolkit (SWT)[6] is a GUI toolkit for the Java platform which is used by Eclipse and Eclipse-based applications. It is an alternative to other Java GUI toolkits such as the Abstract Widget Toolkit AWT or the Swing toolkit. SWT provides widgets such as buttons or text fields, layout managers to arrange widgets, and the infrastructure which is necessary to operate a GUI. It supports several platforms such as Windows, Linux, and Mac OS X.

As described above, the operating system detects user interactions with hardware devices, handles low-level processing, and places GUI events in an application's event queue. SWT uses an event loop to transfer GUI events from the underlying operating system event queue to the SWT event system. This event loop runs continuously during the execution of an SWT application, reads GUI events from the operating system event queue, and dispatches these events, i.e. triggers routines which handle them. SWT event handling is performed in the main thread of an SWT application. When implementing an SWT application, developers have to take care to perform long operations triggered by GUI events in a separate thread to allow the event loop to return quickly and handle the next GUI event. Otherwise, the event loop is blocked and the reaction of the application to user interactions is delayed.
SWT has a pre-defined set of SWT GUI events. For example, the MouseEvent represents mouse related interactions. To get notified when certain SWT GUI events occurr, developers can implement listeners and register them at specific widgets such as buttons or at the SWT event queue. Listeners are defined via interfaces and have to implement specific listener methods which are called when a corresponding SWT GUI event is detected. For example, the MouseListener targets SWT GUI events generated because of mouse interactions. It defines the methods mouseDoubleClick, mouseDown, and mouseUp which represent user interactions of double clicking, pressing a mouse button, or releasing a mouse button, respectively. When a listener method is called, it can perform a specific action or log the event.

Some sensors of the MALTASE framework are implemented as listeners for particular SWT GUI events. When they get activated, they analyze the SWT GUI event and concerned GUI widgets and log this information.

[4]http://help.eclipse.org (Accessed Feb 2015)
[5]http://www.eclipse.org/swt/javadoc.php (Accessed Feb 2015)
[6]www.eclipse.org/swt/ (Accessed Feb 2015)

2.2.6 Software Analytics

Developers produce and manipulate many artifacts during software engineering: source code, requirements documents, models, e-mails, test cases, and many more. Because of open source development, platforms like SourceForge[7] or GitHub[8], and the Internet, many of those artifacts are available to researchers [121]. Driven by the availability of these datasets as well as they availability of algorithms from the fields of machine learning and data mining, the field of software analytics has emerged over the last years [121]. Menzies and Zimmermann define software analytics as "the analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions." [121]. They identified three important success criteria: software analytics must be actionable, relevant, and real-time. Actionable denotes that developers can base their actions on software analytics results, i.e. exploit them in their tasks or in their decisions. Relevant denotes that the result of software analytics must be interesting and appropriate for developers as target audience. And finally, real-time denotes that knowledge gained from analytics is available when needed and that decisions are based on recent, not outdated, data.

This dissertation investigates the acquisition of usage knowledge by monitoring and analyzing user interactions. Hence, it is part of software analytics as it applies analysis algorithms to interaction traces.
Several datasets for software analytics have been made available to researchers in the last years [121]. In contrast, few usage datasets are publicly available, probably because of privacy concerns. For example, only one of 13 repositories of software engineering data listed by Menzies and Zimmermann [121] do provide user input data: Open Hub[9] contains user reviews of hosted projects, i.e. textual user feedback but no monitored usage data. This fact makes software analytics research on usage data difficult because the unavailability of usage datasets usually requires researchers to collect own datasets. An exception are web applications because web servers usually keep a protocol of page requests in the web server log.

[7] http://sourceforge.net/
[8] https://github.com/
[9] https://www.openhub.net/

Chapter 3

Problem Case Study

Different types of information characterize software usage from the user's perspective: the goals and tasks of the user, the interactions of the user with the user interface to accomplish those tasks, the context in which the user employs an application, as well as problems the user is facing. This information constitutes the user perspective of an application, i.e. why and how users use an application. While the user perspective is obviously known to users, developers might not be aware of it.

In this chapter, we present an exploratory case study about software usage. More specifically, the study investigates information needs of developers regarding software usage, whether usage information is available to developers, and how developers exploit such information. The results of the study form the basis for this dissertation, i.e. research efforts described in the dissertation aim to address information needs identified during the study.
A case study is an empirical research method "investigating contemporary phenomena in their context" (Runeson and Höst [157]) and we chose this research method because our goal is to study the state of the practice, i.e. the situation of professional software developers in their work context.
The case study focuses on the activity of program comprehension because it is an import sub-activity of many evolution tasks such as bug fixing or feature implementation. Software evolution - the activity of changing an existing application - requires developers to understand the application before performing changes and such an understanding is the goal of program comprehension.

We re-analyzed a dataset collected in the course of a general, exploratory case study about program comprehension named "PUNGA" (cf. Roehm et al. [154] and Maalej et al. [117]). The purpose of the PUNGA case study was to investigate developer strategies, information needs, and tool usage during program comprehension in real world contexts. Two findings of the PUNGA case study are relevant for this dissertation:

- "The way in which end users use an application is helpful context information in program comprehension. In many cases this information is missing."
  Hypothesis 18 and 19 of Roehm et al. [154]

- "Developers interact with the user interface of the software to test if the application behaves as expected and to find starting points for further inspection."
  Hypothesis 3 of Roehm et al. [154]

We investigate these two findings in more detail and focus on two aspects: First, we study developer information needs regarding software usage. Second, we study

comprehension strategies in which developers put themselves in the role of users by interacting with the user interface.

This chapter is organized as follows: Section 3.1 describes the design of the case study, Section 3.2 presents its results, Section 3.3 discusses related work, and finally Section 3.4 concludes it. During the description of the case study, we follow the guidelines of Runeson and Höst [157].

3.1 Design

In this section we describe research questions, participants, data collection procedures, data analysis procedures, validity measures, and limitations of the case study. As the case study investigates information needs of developers regarding software usage, it is an exploratory case study.

3.1.1 Research Questions

The case study investigates the following research questions:

> **Research Question RQ4:**
> What are the information needs of developers during program comprehension regarding software usage?
> For which of these information needs is the information available to developers?
>
> **Research Question RQ5:**
> When and why do developers put themselves in the role of a user by interacting with the user interface during program comprehension?

RQ 4 aims to identify which types of information about software usage is interesting for and relevant to developers during program comprehension. Further, it investigates whether such information is usually available to developers. RQ 5 explores the finding that developers behave like end users in more detail. It aims to identify situations when developers assume the user role and to discover the motivation of developers for this behavior.

3.1.2 Case and Participant Selection

The case of this case study are software developers from industry companies during program comprehension tasks. Hence, we used the following criteria to select participants for this study: Researchers and students were excluded if they were not simultaneously software developers working in software industry. Participants had to work for a software company and had to spent at least part of their working time on tasks involving reading or changing source code. Further, participants had to work on applications with a user interface. This constraint was necessary because we study information needs regarding software usage. We did not put any constrains

on company size, team size, development methodology, role of participants, or programming language used.
As we re-analyzed an existing dataset, we did not recruit participants but applied these criteria to participants of the PUNGA study and checked if they are met. We had to exclude seven participants of the PUNGA study as they worked on an automotive software system without a user interface.

Table 3.1 provides an overview of the participants. We analyzed the data from 21 participants from six companies. Two companies were located in Spain and four companies in Germany. The domain of developed applications varied and included facility control, fleet management, event management, port management, computer aided design, product management and content management. Most participants had the role of a developer, i.e. mainly implementing new features, or a maintainer, i.e. mainly fixing bugs. Few participants assumed simultaneous roles of developer and manager. Programming languages used included Java, SQL, Delphi, Python, C, VisualBasic, VisualBasic .NET, and C#. Most participants used an IDE such as Eclipse, Netbeans, or VisualStudio. A few used only source code editors such as vi. The development experience of the participants varied from 1.5 to 19 years. We also collected the experience of participants with their current technology, i.e. programming language or application development framework, which ranged from 0.3 to 18 years. During the observation, participants worked on self-selected tasks. Task types included the implementation of a new feature or fixing a bug, familiarization with a new application, reviewing an application for documentation purposes, or porting a feature from one customer-specific version of the application to another version. The number of tasks participants completed within the observation time of 45 min varied: while some participants could not finish one task in this time, others completed three tasks because of the varying complexity of the tasks.

3.1.3 Data Collection Procedures

The PUNGA study used a combination of observation and interview to collect data because both research methods complement each other: An observation can reliably reveal when and how developers exploit software usage information, e.g. how they reproduce failures based on reproduction steps described in a bug report. But it is not possible to elicit the motivation behind an observation or judge whether an observation is representative for the normal behavior of participants. Such information can be uncovered by a discussion between participant and observer during an interview. Further, interviews allow participants to identify missing information which cannot be observed easily.

Each observation took place for 45 minutes, immediately followed by an interview. The interview lasted for another 45 min to keep the whole session at a length of 1,5 h. In each session, one researcher observed and interviewed one participant. Overall, two researchers conducted the study sessions with the author being one of them.

Table 3.1: Overview of Case Study Participants (Adapted From [154])

Id	*Company*	*Role*	*Application Domain*	*Technology and Tools Used*	*Dev. Exp. (y)*	*Tech. Exp. (y)*	*Task(s) During Observation*
P1	C1 (DE)	Developer, Maintainer	Facility control	Java, Netbeans	4.5	4.5	Application familiarization
P2	C2 (DE)	Documenter	Fleet management	PL/ SQL, Oracle SQL Developer	3	0.3	Documentation
P3	C3 (ES)	Manager, Developer	Event management	Delphi, Delphi IDE (Client), Java, Eclipse (Server)	8	6	Bug fixing
P4	C3 (ES)	Manager, Developer	Event management	Delphi, Delphi IDE (Client), Java, Eclipse (Server)	8	6	Feature impl.
P5	C4 (ES)	Developer	Port management	Oracle DB, Java, Oracle Toad, Eclipse, Tomcat	2	2	Bug fixing Feature impl.
P6	C4 (ES)	Maintainer	Port management	Oracle DB, Java, Oracle Toad, Eclipse, Tomcat	1.5	1	Feature impl. (2x)
P7	C4 (ES)	Developer, Maintainer	Port management	Oracle DB, Java, Oracle Toad, Eclipse	4.5	2.5	Porting feature
P8	C5 (DE)	Developer	Computer Aided Design	Python, Eclipse	1.5	1.5	Feature impl. (3x)
P9	C5 (DE)	Developer	Computer Aided Design	Python, Eclipse	19	3	Bug fixing
P10	C5 (DE)	Developer	Product management	Python, SQLite, Toad	5	5	Code review
P11	C5 (DE)	Developer	Databases	C, Python, XML, Vi	7.5	5	Feature impl. Porting feature
P12	C6 (DE)	Developer	Content management	C#, Visual Studio	3	3	Bug fixing
P13	C6 (DE)	Developer	Content management	VB, VB .NET, Visual Studio	11	8	Bug fixing
P14	C6 (DE)	Developer	Content management	VB .NET, C#, Visual Studio	11	8	Bug fixing
P15	C6 (DE)	Developer	Content management	Java, Tomcat, NetBeans	16	11	Feature impl.
P16	C6 (DE)	Developer	Content management	Java, NetBeans	0.5	1.5	Feature impl.
P17	C6 (DE)	Developer	Content management	Java, Eclipse	11	11	Feature impl.
P18	C6 (DE)	Developer	Content management	VB .NET, SQL Server, Visual Studio	3	3	Bug fixing
P19	C6 (DE)	Developer	Content management	VB, VB .NET, Visual Studio	18	18	Bug fixing
P20	C6 (DE)	Developer	Content management	VB. NET, Visual Studio, Editor	8	4	Bug fixing
P21	C6 (DE)	Developer	Content management	VB .NET, Visual Studio	10	5	Bug fixing

Table 3.2: Excerpt from Observation Protocol of Participant P5 (Source: [154])

Daytime	*Relative time*	*Observation/ Quote*	*Postponed questions*
...	...	...	...
10:19	00:27	Read Jira ticket *Comment: "this sounds like the ticket from yesterday"*	
10:20	00:28	Refresh source code repository	
10:24	00:32	Publish code to local Tomcat	
10:26	00:34	Debug code in local Tomcat	*Why debugging?*
10:28	00:36	Open web application in browser and enter text in text fields	
10:29	00:37	Change configuration in XML file content.xml *Exclamation: "not this complicated xml file again"*	*How known what to change?*
10:30	00:38	Publish changes to local Tomcat	
10:31	00:39	Debug local Tomcat	
...	...	...	...

Observation

The purpose of the observation was to answer questions about developer behavior and information accesses such as "Which information do participants access during program comprehension?" or "How do participants exploit certain types of information?". The observations were conducted with the think aloud method (Ericsson and Simon [47]). At the beginning of an observation session, participants were introduced to the goal and procedure of the study. Then, participants were asked to work on their normal tasks but comment on what they are doing. This enables the observing researcher to understand participant behavior in more detail. If a participant stopped commenting, the researcher asked a question to re-trigger the comments. Care was taken not to interrupt participants during their work and questions which required discussion were deferred to the following interview. The participants worked on self-chosen tasks from their normal task list. We put the constraint on the task that it should require a minimal amount of program comprehension, i.e. to comprehend an existing piece of software.

Each observation session lasted for about 45 minutes and was conducted by one researcher who observed one participant. The observation sessions were not recorded because of confidentiality problems and to minimize the influence of being observed on participant behavior. Instead, the observing researcher kept notes of important events, quotes, and observations in an observation protocol. Table 3.2 shows an excerpt of such an observation protocol. To document as much information as possible, the observation protocol created during an observation session was revised and extended by the observing researcher on the same day.

Interview

The purpose of the interview was to clarify observations from the observation session, to answer questions about the motivation behind observed behavior, and to collect information about non-observable issues, e.g. "Why did you do X?" or "Which information is usually missing?". A semi-structured interview approach with two phases was used: In the first phase, interesting but unclear observations from the observation session were discussed, e.g. "How did you know that the bug was in method X?". In the second phase, open questions about comprehension strategies and missing information during program comprehension were asked. If a participant gave an interesting but unclear answer, it was discussed in more detail. A questionnaire[1] with seven open questions helped the observing researcher stay focused. The observing researcher kept interview minutes and revised and extended them on the same day.

Testing of Data Collection Procedures

The template for the observation protocol and the questionnaire was tested in a dry run with a graduate student under study conditions. While the template for the observation protocol proved usable, the questionnaire contained too many questions for 45 minutes. Hence, some less important questions were removed and a couple of similar questions were merged.

3.1.4 Analysis Procedures

We analyzed the observation protocols and interview minutes originating from study sessions with 21 participants. The data was collected in the course of the PUNGA study. Overall, the observation protocols contained 1.665 items such as events, observations, or quotes (79 items per participant on average). The interview minutes contained 139 answers (7 per participant on average), excluding questions about participants themselves. The analysis was performed by the author. In case of uncertainties in protocols and minutes from sessions where the author did not participate, the uncertainty was discussed with the researcher who conducted the session.

The analysis was performed in three phases. In the first phase, we established a coding scheme. We read all observation protocols and interview minutes and created lists for the following items: observed or reported information needs regarding software usage and situations in which participants interacted with the user interface of the application. We established a code for each item of these lists. For example, we found different purposes why developers interact with the user interface: to reproduce a failure (code FAILURE REPRO), to find source code (code FIND CODE), or to test a change (code UI TEST). In the second phase, we coded the data. We reanalyzed all observation protocols and interview minutes and labeled all instances of an item with its code. Finally we used the codes to establish the number of participants, the number of companies, and the number of observers for each code. In the third phase, we created textual descriptions to summarize all instances of a particular code. These descriptions are presented in Section 3.2. Furthermore, we

[1] https://sites.google.com/site/pungaproject/templates (Accessed Jan 2015)

condensed each description to a one-sentence finding which is presented together with the textual description. Those findings summarize our results in form of a catalogue of findings.

3.1.5 Reliability and Validity Measures

We used the following measures to ascertain the reliability and validity of results:

- **Minimum Participant Support**
  All results reported have been observed or reported by at least two participants. This ensures that the results have a "minimal generalizability" and excludes results based on individual factors.

- **Participant Checking** (Creswell [35])
  We sent the catalogue of findings to the participants and asked them whether they agree with each finding using a five item Likert scale from "Strongly disagree" to "Strongly agree". Two participants responded to this call. Participant P2 strongly agreed or agreed with 9 findings (F1-F2, F4-F7, F9-F10, F12), was undecided for 5 findings (F8, F11, F13-F15), and disagreed with finding F3, commenting that in his perspective information about user goals and user needs was too high level to be used in program comprehension. Participant P6 strongly agreed or agreed with 13 findings (F1-F13) and was undecided for findings F14 and F15, commenting that it depends on the context.

- **Data Triangulation** (Creswell [35])
  As we analyzed data which were collected in observations and interviews, we triangulate between these data sources. A result that was both observed and reported increased its validity.

- **Independent Peer Observations** (Rosnow and Rosenthal [156])
  Sessions of the PUNGA study were conducted by two researchers at different companies. Hence, we triangulate between observers to address observer bias. A result that was observed or reported by two observers independently increased its validity. We report for each result the number of supporting observers. Most of the results are supported by two observers.

3.1.6 Threats to Validity

As every empirical study, the design of the case study has threats to validity which we discuss in this section.

Data analysis was performed by a single researcher, the author. This introduces the threat that results may be biased by assumptions and expectations of the researcher. We took several measures to mitigate this threat: In case of uncertainties about data from sessions which the author did not participate, those were discussed with the researcher who conducted the corresponding session. Further, the other researcher who conducted sessions during the PUNGA study read a draft of this chapter and did not find mismatches with her experiences. Also, we performed participant checking in which participants evaluated the correctness of the findings.

And finally, we took care to only report results which did not require a lot of interpretation, e.g. actions of participant clearly documented in the observation protocol or information needs clearly reported in interview answers. Please note that the case study was the first part of our dissertation research and inspired the MALTASE framework.

As there is no clear definition of the population of software developers, the representativeness of the study sample is unknown and the generalizability of the study results remains open. Because we studied 21 participants in different companies, in different team, in different project roles, performing different comprehension tasks, using different development methodologies, and using programming languages, we argue that the study results have some of external validity but future work should address the issue of generalizability.

Participants had to work with an application with a user interface. While this is reasonable because we are studying software usage by users, it constraints the applicability of the study results: They are applicable to applications with a user interface, not to all types of applications.

As we analyzed an existing dataset, data was not collected with our specific research questions in mind. Consequently, some observations or questions relevant to our research questions might have been ignored by the observers and not appear in the dataset. The focus of the PUNGA study was broad and explorative and this case study investigates sub-aspects of the PUNGA study: comprehension strategies involving information about software usage instead of comprehension strategies in general and information needs regarding software usage instead of information needs in general. Hence, we argue that the results are valid but might be incomplete.

As the observations of 45 minutes length covered only a fraction of participants' work time, relevant observations might have been missed. We argue that extending the duration of observation sessions would not change the results dramatically because the tasks were self-chosen and 21 participants in different contexts were studied. Furthermore, this limitation concerns the completeness of the results but not their correctness.

Some degree of subjectivity is involved when describing aggregated results and phrasing findings. To cope with this limitation, we report many direct quotes and observations which support a certain result. Nevertheless, care should be taken when interpreting the results.

Half of the participants worked for company C6. Hence, the results may be skewed towards this company. C6 is a large international software company that develops enterprise information management systems. Study sessions were conducted in their German office which has approx. 600 employees.

3.2 Results

Table 3.3 summarizes the results of the case study. This section describes the results in detail and lists corresponding findings.

Table 3.3: Overview of Case Study Results
Frequencies Indicate Number of Participants, Companies, and Observers.

Result	*#Part.*	*#Co.*	*#Obs.*
Information Needs of Developers Regarding Software Usage (RQ 4)			
(IN1) Use cases and user behavior	6	4	2
(IN2) User goals and user motivation	3	3	2
(IN3) Failure reproduction steps	2	2	2
(IN4) Application domain concepts	2	2	1
(IN5) UI-focused documentation style	2	1	1
(IN6) Availability of information about software usage	(2)	(1)	(1)
Developers in the Role of Users (RQ 5)			
(UR1) Developers interact with user interface to ...			
... reproduce a failure	9	2	2
... find relevant source code	5	3	2
... test an implemented change	5	2	1
... trigger the debugger	5	1	1
... familiarize with unknown part of application	3	3	2
(UR2) Developers conceptually map UI elements to ...			
... source code	6	4	2
... data (data structure/ data flow)	2	2	2
... algorithms (algorithm steps/ control flow)	2	2	1

3.2.1 Information Needs of Developers Regarding Software Usage

This section describes results regarding information about software usage as answers to RQ 4. As such information is rarely available to developers, most results originate from the interviews.

(IN1) Information about Use Cases and User Behavior

Six participants from four companies reported that information about use cases and user behavior is necessary or helpful during program comprehension. Participant P2 reported that it is unclear for him "how a user uses the software" but he needs this information. Similarly, participant P8 reported that he is "missing information about use cases, i.e. how the users want to use a software product" and participant P11 commented that "not all use cases are documented". Participant P10 mentioned that information "how the software is used" is helpful during program comprehension.

This result matches related work by Begel and Zimmermann [15] who found that "How do users typically use my application?" is an important developer question.

> **Finding F1:**
> Information about use cases "performed" by users and user behavior is helpful information for developers during program comprehension.

Finding F2:
Information about use cases "performed" by users and user behavior is rarely available to developers during program comprehension.

(IN2) Information about User Goals and User Needs

User goal and user need information is more abstract than use case and user behavior information (cf. IN1). Information about goals and needs describes why a user employs an application while information about use cases and behavior describes how users employ the application. Hence, we distinguish between both types of information.

Three participants from three companies reported that information about user goals and user motivation is necessary or helpful during program comprehension. Participant P2 used a self-established scheme to document software functionality which consisted of 14 items. One item was "user goals" and the participant commented that "'user goals' describes goals that the user intends to achieve when executing this function". He also mentioned that it is often unclear to him how users employ the application to achieve a certain goal. Participant P3 reported that "the purpose of an application ('Why is it developed?')" and "needs that are supported by the application" is important information to comprehend an application. Similarly, participant P14 commented that "the business cases where customers describe what they want to have and what they want to do with the application" is helpful information during program comprehension.

This information need is reflected when developers document software requirements with user stories: Templates for user stories (e.g. "As a <type of user>, I want <some goal> so that <some reason>" by Mike Cohn[2]) require developers to document the type of user, the user goal, and the reason behind each user story.

Finding F3:
Information about user goals and user needs is helpful information for developers during program comprehension.

(IN3) Information about Failure Reproduction Steps

Two participants from two companies reported that information about failure reproduction steps is necessary or helpful during program comprehension. We argue that reproduction steps are one type of usage information because reproduction steps of GUI applications usually consist of the user actions preceding a failure. Participant P3 reported that "steps to reproduce the bug, i.e. what the user did" is necessary information to fix bugs. Participant P21 reported that failure reproduction is a difficult activity because "the steps are not described completely" .

This result matches studies by Zimmermann et al. [203] and Laukkanen et al. [105] who found that reproduction steps are often missing or incomplete in bug reports submitted by users, making failure reproduction difficult. Further, Maalej et al. [117]

[2]http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template (Accessed December 2014)

found that 93 % of developers have difficulties to reproduce failures because of missing information at least weekly.

Finding F4:
Information about failure reproduction steps is helpful information for developers during program comprehension and bug fixing.

(IN4) Information about Application Domain Concepts

Two participants from two companies reported that information about the application domain is necessary or helpful during program comprehension. Participant P2 reported that "a description of the most important concepts of the application domain" would be helpful. Currently he asks an expert working for the customer to provide such information if necessary. Participant P3 reported that he is missing information about "the business of the customer".

This result matches related work by Shaft and Vessey [164] who found that knowledge of application domain concepts plays an important role in program comprehension and that developers use different comprehension processes depending on their familiarity with the application domain.

Finding F5:
Information about application domain concepts is helpful information for developers during program comprehension.

(IN5) UI-focused Documentation Style

We observed an interesting style of documentation for two participants from the same company. Participants P6 and P7 worked "UI-driven", i.e. they used it as starting point to locate errors or understand functionality. This UI-driven comprehension approach fits to the documentation style in their company: Software functionality is documented in the form of use cases with screenshots, i.e. each use case is documented by a set of screenshots of the UI views which are relevant for the use case. Additionally, database tables and database fields related to the use case are also documented.
This documentation style shows the importance of information about use cases during program comprehension (see IN1) and illustrates a UI-driven comprehension approach where developers interact with the user interface as starting point for comprehension (see UR1).

(IN6) Availability of Information about Software Usage

In contrast to reports of developers about their information needs regarding software usage, we did not observe any participant exploiting information about software usage - besides the UI-focused documentation style described in IN5. Hence, we hypothesize that information about software usage is rarely available to developers during program comprehension.
This finding has one limitation: Ten participants worked on a bug fixing task and

many of them examined a bug report. The dataset does not indicate if those bug reports were submitted by users and contained reproduction steps. Hence, it might be the case that some participants exploited information about software usage, namely user actions preceding the failure. But this is unlikely as studies ([105, 203]) have shown that information about reproduction steps is often missing in bug reports submitted by users.

This result matches related work by Maalej et al [115] who found that communication gaps between developers and users lead to ignorance of developers about software usage.

> **Finding F6:**
> Information about software usage from the user perspective, i.e. why and how users employ an application, is rarely available to developers during program comprehension.

3.2.2 Developers in the Role of Users

One goal of this case study is to investigate when developers interact with the user interface of the application during program comprehension (RQ 5), putting themselves in the role of a user. We present the results in this section. More specifically, we describe five purposes why developers interact with the user interface, three conceptual mappings developers perform between UI elements and software internals, and two unexpected interactions of developers with the user interface.

Developers Interacting with UI to Reproduce Failures (UR1.1)

Nine participants from two companies interacted with the user interface to reproduce a failure or reported to do so. Participant P3 described a three step strategy to fix a bug: "Step 1: Try to reproduce bug, Step 2: Identify error on user interface, Step 3: Find related code". Participant P13 worked on a bug report which described a bug in the logic of a report generator. His first action was to start the application and trigger the report generation logic from the user interface to reproduce the failure. Participant P21 tried to reproduce a bug caused by a certain configuration setup. He repeatedly changed configuration parameters in the user interface and restarted the application until he identified a specific, failure-inducing configuration. Another approach for failure reproduction was reported by participant P4: He reported that he "tries the application in different browsers to rule out the browser as the source of a failure".

> **Finding F7:**
> During program comprehension, developers of an application with a user interface interact with the user interface to reproduce failures.

Developers Interacting with UI to Identify Relevant Source Code (UR1.2)

Five participants from three companies interacted with the user interface to locate source code which is relevant for their current task or reported to do so. Participant

P12 repeatedly changed the local language setting in the user interface to identify which code method was triggered by this action. Participant P15 searched for the code which is executed upon a button click. Participant P17 used text search to search for text labels of GUI elements within source code.
This result matches with work by Ko et al. [97] who found that the first step during program comprehension is to search for relevant code. While Ko et al. observed manual code searches and code searches using search tools, we found a complementary strategy: interacting with the UI and mapping UI elements to code. The behavior of participant P17 matches to the developer question "Where in the code is the text in this error message or UI element?" identified by Sillito et al. [168].

Finding F8:
During program comprehension, developers of an application with a user interface interact with the user interface to locate source code which is relevant to their current task.

Developers Interacting with UI to Test an Implemented Change (UR1.3)

Five participants from two companies interacted with the user interface to test a change they had implemented before or reported to do so. Participant P5 applied the following working strategy: change code, recompile, deploy, start the application, interact with the user interface to check whether the change is correct. He repeated this strategy several times. Several participants fixed a bug and checked the correctness of the fix by interacting with the user interface to ensure that the failure did not occur anymore.

Finding F9:
During program comprehension, developers of an application with a user interface interact with the user interface to test implemented changes.

Developers Interacting with UI to Trigger the Debugger (UR1.4)

Five participants from the same company interacted with the user interface as starting point for debugging or reported to do so. Participant P21 worked on a bug which occurred in a certain dialog window of the user interface. He set a breakpoint to the code of this dialog, started the application in debug mode, and interacted with the user interface to trigger the buggy dialog. Now the debugger was triggered and he started step-by-step debugging. All five participants worked in a similar way: set one or more breakpoints to code locations considered relevant, start the application in debug mode, interact with the user interface until a breakpoint is reached and the debugger becomes active, and debug the application.
Activating the debugger by interacting with the user interface is similar to interacting with the user interface to find relevant code (cf. UR1.2). Debugging can be considered as one possible approach to find relevant code which requires developers to make assumptions about relevant code by placing breakpoints. But relevant source code can also be searched without debugging.

Finding F10:
During program comprehension, developers of an application with a user interface interact with the user interface to trigger the debugger, i.e. till a breakpoint is reached and the debugger is activated for inspection of application state or step-by-step debugging.

Developers Interacting with UI to Familiarize With Unknown Part of Application (UR1.5)

Three participants from the three companies interacted with the user interface to familiarize with an unknown application or unknown part of the application or reported to do so. Participant P1 reported his strategy to familiarize with a new, unknown application: "I start the application and examine what the user can do or click. Then I look at the code which is called upon clicks on buttons." Participant P2 had to familiarize with an unknown application which performed different types of calculations and executed the application to inspect its functionality. He entered parameter values in the user interface and ran the calculation with different parameter values to understand the calculations performed.

Finding F11:
Developers of an application with a user interface interact with the user interface to familiarize with an unknown part of the application.

UI Interaction Patterns of Developers

We described five purposes why developers interact with the user interface during program comprehension. These interactions did not occur in isolation but also in combination with each other. In particular, we identified two strategies for bug fixing tasks. The first strategy contains of four steps: interact with the user interface to reproduce the failure (step 1), interact with the user interface to start debugging (step 2), change code identified during debugging to fix the bug (step 3), and finally interact with the user interface to test the fix (step 4). The second strategy is a sub strategy without the debugging step, i.e. it consists of steps 1, 3 and 4.

Finding F12:
A strategy to fix a bug in an application with a user interface is to reproduce the bug in the UI, trigger the debugger by interacting with the UI, change code which was identified during debugging, and test the correctness of the fix by interacting with the UI.

During the observations using the think aloud method, it became clear that some developers mapped elements of the user interface to internal elements of the application. We found three different types of such mappings.

Mapping of UI Elements to Source Code (UR2.1)

Six participants from four companies mapped user interface elements to source code or reported to do so. Participants P1 and P15 mapped interactions with UI widgets such as button clicks to code triggered by the interaction. Participant P12

changed the local language setting in the user interface to identify the code method triggered by this action. Participants P2 and P17 mapped labels of text fields in the user interface to variables in the code. For example, P2 familiarized with an unknown application which performed calculations with several input parameters. He searched for the the names of those parameters in code and thereby mapped parameter names from the user interface to the corresponding variables in code.

Finding F13:
During program comprehension, developers of an application with a user interface conceptually map elements of the user interface to source code.

Mapping of UI Elements to Data (UR2.2)

Two participants from two companies mapped user interface elements to data structures or reported to do so. Participant P4 mapped the values of text fields and other GUI widgets to the contents of the database. Because of a mismatch between the data shown in the user interface and the database (there were less data items in the database than in the GUI) he detected that he is using the wrong database server for testing. He also tested the the correctness of data storage logic by entering text in text fields and checking if they arrive correctly in the database. Participant P10 explored an application during a code review and found that data entities shown in the GUI are stored in a data dictionary.
This finding matches the systematic comprehension strategy identified by Littmann et al. [110]: "The programmer using the systematic strategy traces data flow through the program to understand global program behavior".

Finding F14:
During program comprehension, developers of an application with a user interface conceptually map elements of the user interface to data flow within or data structures of the application.

Mapping of UI Elements to Algorithm Execution (UR2.3)

Two participants from two companies mapped UI elements to the execution of an algorithm or reported to do so. Participant P2 started the application and repeatedly changed input parameters of a calculation algorithm via the user interface. He executed the calculation algorithm with different parameter values to understand the calculation performed. This behavior targets the functionality of an algorithm and resembles a black box approach as it considers just the input and output values of the algorithm. Participant P3 related the control flow of a processing routine to elements of the user interface: "we only passed two times in this loop because we have two categories of events in the UI" (two categories of events were visible in the user interface of a calendar application). This behavior targets control flow of an algorithm and resembles a white box approach as it investigates the steps of the algorithm.
The difference of mapping UI elements to algorithm execution compared with mapping to source code is that this mapping focuses on the steps of the algorithm, not

the code which implements those steps. This result can be described as "algorithm reengineering", i.e. a developer infers the steps of an unknown but implemented algorithm from application behavior visible in the user interface.

> **Finding F15:**
> During program comprehension, developers of an application with a user interface conceptually map elements of the user interface to the control flow of algorithms implemented by the application.

Unexpected Interactions of Developers with UI

During the observations, we found two surprising and unexpected examples of a developer interacting with the user interface: Participant P16 worked on the task to integrate results of several search engines into one common result set. As this typically is a backend task, we did not expect the participant to deal with the user interface. However, the participant created a GUI project to test his implementation. Similarly, participant P10 performed a code review to investigate how the application could be extended to support a new concept of the application domain. In this case, the participant examined the user interface before analyzing code.

3.3 Related Work

This section reviews related work in two areas: information needs of developers and empirical studies about comprehension strategies of developers.

Information Needs of Developers

Several researchers studied information needs of developers during software development and software evolution. Begel and Zimmermann [15] collected a list of questions which are important for developers and identified 145 questions in 12 categories. One of their categories, "customers and requirements", contains questions about software usage. They also asked developers to prioritize the 145 identified questions. Interestingly, the two top rated questions were usage-related: "How do users typically use my application?" and "What parts of a software product are most used and/or loved by customers?". Similarly, Buse and Zimmermann [26] present information needs during software analytics. They describe the decision scenario "understanding customers" which "leverages information about customer behavior when making decisions", e.g. to "understand how a user is using our product" or whether users are "performing tasks we expect". Results of these two studies match our finding that developers are interested in information about software usage and that such information is important. Our results complement their results and identify additional aspects of software usage which are interesting for developers. Maalej et al. [117] investigate the frequency of different information needs in a survey among 1477 developers. Besides the need for information about reproduction steps for bugs (which was the most frequent information need and encountered at least weekly by 93 % of respondents), they do not investigate usage or user information. Sillito et al. [168] observed 25 developers and compiled a list of 44 questions programmers

ask during code changes. They explicitly focus on questions about the code base and do not target questions about users or usage. They also observed participants mapping UI elements to application internals. Ko et al. [96] observed 17 developers at Microsoft and established a list of 21 information needs. They do not discuss information needs regarding usage or user, but also found that bug reproduction steps are nearly impossible to acquire. Fritz and Murphy [54] interviewed 11 software developers and identified 78 questions of developers. Those questions consider mainly code issues and project information and don't target usage or user. Begel et al. [14] surveyed 110 developers at Microsoft and identified the ten most important information needs regarding inter-team coordination among developers. They do not target usage and user information needs. Breu et al. [22] study information needs in bug reports by analyzing 600 bug reports of two open source projects. They found that "the role of users goes beyond simply reporting bugs: their active and ongoing participation is important for making progress on the bugs they report". Further, they found that developers frequently request the following information from users: reproduction steps, information about the environment, stack traces, and screenshots. While their results match with our result that developers are interested in reproduction steps, our study is broader as it does not only consider information needs during bug fixing.
Summarizing, the importance as well as the frequent unavailability of failure reproduction steps was identified by several related works. Besides reproduction steps, most related studies do not investigate information needs regarding usage or user. Hence, this case study complements them by investigating information needs regarding software usage from the user perspective.

Empirical Studies About Comprehension Strategies

Several empirical studies have been conducted to investigate strategies of developers during program comprehension, i.e. which activities developers perform during program comprehension and how they combine different activities to perform a certain task. Ko and Uttl [99] study individual differences of developers comprehending an unfamiliar application. They identified "Use of GUI" as a comprehension strategy which "uses GUIs to comprehend programs" but do not describe or discuss this behavior. Ko et al. [97] also study developers understanding unfamiliar code. They identify the developer action "Testing Paint by executing it from Eclipse" and observed that "developers spent a tenth of their time testing the Paint application". But they do not describe how this test was conducted and how developers interacted with the GUI. LaToza et al. [104] study developer tools, activities, and practices with a focus on activities targeting source code. Their example of a bug investigation task also describes a developer who first reproduces a failure by interacting with the user interface and then triggers debugging by interacting with the user interface. Singer et al. [169] study daily activities of software engineers. Koenemann and Robertson [100] study comprehension strategies of expert developers. Corritore and Wiedenbeck [34] compare comprehension strategies used by developers using object-oriented and procedural programming languages. Vessey [193] investigates differences in debugging behavior of expert and novice developers. Sillito et al. [167] observe nine developers changing unfamiliar code using pair programming. Robillard

et al. [145] study comprehension behavior of five developers during code changing tasks and identify factors related to the effectiveness of program investigation.

Summarizing, the empirical studies discussed above focus mainly on source code analysis or debugging. Some of them mention developers interacting with the user interface but do not describe or analyze this behavior in depth. Hence, this case study complements them by analyzing developer interactions with the user interface during program comprehension.

3.4 Discussion

This section presents conclusions from the case study and discusses implications and future work for researchers, practitioners, and tool vendors. Limitations of the case study are discussed in Section 3.1.

3.4.1 Conclusions

Case study results about information needs (see IN1-IN4) show that developers are interested in information about software usage from the user perspective because they consider such information helpful or necessary during program comprehension. More specifically, developers are interested in use cases and user behavior, user goals and user needs, failure reproduction steps, and application domain concepts. These information needs complement information needs identified by related work.
However, we observed only two participants (see IN5) accessing and exploiting usage information during program comprehension. This reveals a mismatch between information needs and information availability. Related research already identified two reasons behind this mismatch: communication gaps between developers and researchers (Maalej et al. [115]) as well as a perspective change - transiting from being primarily a user to being primarily a developer - of students during their computer science education (Schinzel [159]). Future research is necessary to investigate the mismatch in more detail, especially to investigate reasons behind it and to establish its generalizability. This mismatch probably indicates a way to improve current program comprehension practices by collecting usage information and providing it to developers during program comprehension.

We found five reasons why developers interact with the user interface of the application during program comprehension (see UR1): to reproduce failures, to find relevant source code, to test an implemented change, to trigger the debugger, and to familiarize with an unknown parts of the application. We call this result "UI-based comprehension" because developers acquire information about the application and its behavior by interacting with the user interface, effectively putting themselves in the role of a user. We argue that "UI-based comprehension" denotes a comprehension activity which is a part of a larger comprehension strategy together with activies such as debugging or reading code.
Related work (see Section 3.3) characterizes comprehension strategies as opportunistic (only investigate and understand the part of a program necessary for the current task) or systematic (investigate and understand the whole program). We argue that

the "UI-based comprehension" can be employed by developers in both ways: Systematically by exploring all user interface elements to obtain an overview of the user interface and the features available to users, e.g. when familiarizing with a new application. And opportunistic by exploring only user interface elements necessary for the current task, e.g. when reproducing a failure.

3.4.2 Implications And Future Work

This section presents implications and future work for researchers, practitioners, and tool vendors.

Implications And Future Work For Researchers Based on the study results, we encourage researchers to investigate how users can be involved during software evolution to tackle the mismatch between developer interest in information about software usage and its unavailability to developers. More specifically, they should study how information about user needs and application usage can be acquired, managed, presented to developers, and exploited during program comprehension and software evolution. Further, they should study how the availability of this information impacts program comprehension and software evolution tasks. To this end, Pagano [135] proposes an approach to collect, process, and exploit textual user feedback. This approach requires users to document and share feedback proactively.

Researchers should consider "UI-based comprehension" in future research about program comprehension and investigate it in more detail. Future research directions are the motivation behind the "UI-based comprehension" and the influence of comprehension context (characteristics of individual developer, type of task, type of application) on its use. Additionally, researchers should investigate how developers combine "UI-based comprehension" with other comprehension activities such as debugging or reading source code. Another area of future research is to investigate the conceptual mappings between UI elements and application internals in more detail.

This case study was explorative and resulted in a catalogue of findings. Future work should investigate the generalizability of those findings and contextualize them, i.e. determine in which contexts they are relevant.

Information needs of developers during software development and software evolution has been studied by several researchers in the last years (see Section 3.3). Like this case study, each of these research efforts resulted in a list of information needs. We see future work for researchers in prioritizing and contextualizing these information needs, i.e. to investigate which information need is more important than others and to determine the context in which an information need is relevant. Such an effort could lead to a generally accepted list of information needs together with a description of the context in which each information need is relevant. Dimensions of such a context could be the type of task, the type of application, the experience of the developer, or the familiarity of the developer with the application.

Furthermore, we propose to study "UI concept location" or "UI feature location" which - in analogy to concept or feature location in source code - denote the mapping of application domain concepts or application features to UI elements.

Implications And Future Work For Practitioners Based on the study results, we encourage practitioners to reconsider their strategy for involving users in their evolution process and for eliciting feedback from them. More specifically, to identify potential user involvement channels, to develop approaches to capture and analyze user feedback, to identify ways how to exploit knowledge gained, and to weigh costs and benefits of such approaches. Further, we encourage practitioners to reconsider the comprehension strategies they use in their daily work and to identify comprehension strategies which optimize productivity in their context. Because two participants used a UI-focused documentation style (see IN5), we encourage practitioners to test whether user manuals can be exploited as information source during program comprehension.

Implications and Future Work For Tool Vendors We encourage vendors of software development and program comprehension tools should analyze the UI-based comprehension strategy and consider it in the future development of their tools, i.e. investigate tool support for the UI-based comprehension strategy. For example, to enable developers to debug an application on user interface level instead of source code level by allowing developers to set breakpoints in the UI, to inspect the state of the UI, and to easily navigate from UI elements to application source code. Apple recently released the "View Debugger" feature of its XCode IDE which heads in this direction[3]: it allows developers to pause a running application, to inspect the properties of the user interface, and to jump to relevant source code. Furthermore, we encourage tool vendors to study how conceptual mappings between UI elements and software internals can be supported by tools.

3.5 Chapter Summary

This chapter presented an exploratory case study about information needs of developers during software evolution. The case study focused on information needs about software usage from a user perspective, i.e. why and how users use an application. We observed and interviewed 21 developers from six software companies during program comprehension tasks, mainly bug fixing and feature implementation. We found that developers are interested in use cases and user behavior, user goals and user needs, failure reproduction steps, and application domain concepts. But such information is rarely available to them during software evolution. These results complement related work about developer information needs. Further, the mismatch between developer interest in usage information and its rare availability indicates a potential to improve software evolution practices by collecting such information and providing it to developers. Furthermore, we found that developers interact with the user interface of the target application to reproduce failures, to find relevant source code, to test an implemented change, to trigger the debugger, and to familiarize with an unknown part of the application. By interacting with the user interface, developers put themselves in the role of users. We call this result "UI-

[3]Source: https://developer.apple.com/library/prerelease/ios/documentation/DeveloperTools/Conceptual/WhatsNewXcode/Articles/xcode_6_0.html (Accessed Jul 2014)

based comprehension" and argue that it is part of a broader comprehension strategy together with other comprehension activities like reading source code or debugging. As this developer behavior has not been studied in detail, we suggest to investigate it in more detail and design tools supporting developers.

This dissertation contributes to the effort to tackle the mismatch between developer interest in usage information and its unavailability. It focuses on one aspect of software usage: the interactions of users with the user interface. As those interactions can be captured and analyzed automatically, they constitute an automated feedback channel from users to developers which does not require any effort from users and only few effort from developers. We present the MALTASE framework to capture and analyze user interactions automatically. Further, we evaluate its impact on failure reproduction and user skill analytics. Thereby we address the problem of missing failure reproduction steps which was found in the case study as well as reported in related work.

Chapter 4

The MALTASE Framework

This chapter describes the MALTASE framework for collection, storage, and analysis of user interactions. MALTASE is a shorthand for "Monitoring, Analysis, and ExpLoiTation of User InterActions in Software Evolution". Section 4.1 sketches an overview over the whole framework, Section 4.2 lists framework requirements, Section 4.3 describes a model of user interactions and other relevant concepts, Section 4.4 sketches the framework architecture, Section 4.5 discusses usage scenarios for the framework, and Section 4.6 summarizes the state of research and practice regarding usage monitoring and analysis of usage data.

4.1 Framework Overview

The MALTASE framework consists of four layers: Monitoring and Information Extraction layer, Storage and Transfer layer, Processing and Analysis layer, and Presentation and Integration layer (see Figure 4.1 for an overview). The *Monitoring and Information Extraction layer* implements functionality to detect user interactions at a high level of abstraction and elicit information about detected interactions and their context. The *Storage and Transfer layer* implements functionality to represent monitored data, store them, and transfer them from a user device to a server at the developer site. Data storage denotes storing monitored data temporarily on the user device and permanently on a central server. Transfer denotes the transfer of data between the user device and the developer server. The *Processing and Analysis layer* implements functionality to analyze monitored data. The type of analysis performed depends on the framework application, i.e. the type of information which should be extracted from monitored data. Hence, this layer can be regarded as a toolbox of functionality which developers can use when implementing a specific analysis. The toolbox contains tools such as aggregation of low level data, filtering of data, pattern detection, or classification of interaction sequences or users. Finally, the *Presentation and Integration layer* implements functionality to represent and visualize monitored data and analysis results. Furthermore, it provides functionality to integrate this information in developers tools such as bug repositories or IDEs. This helps developers to analyze monitored data, gain knowledge about usage of the target application, and exploit this knowledge in their software evolution tasks without the need for switching tools.

The details of the MALTASE framework are defined by its requirements, its model, and its architecture which are described in Section 4.2, Section 4.3, and Section 4.4. An empirical evaluation of the MALTASE framework is presented in Chapter 6.

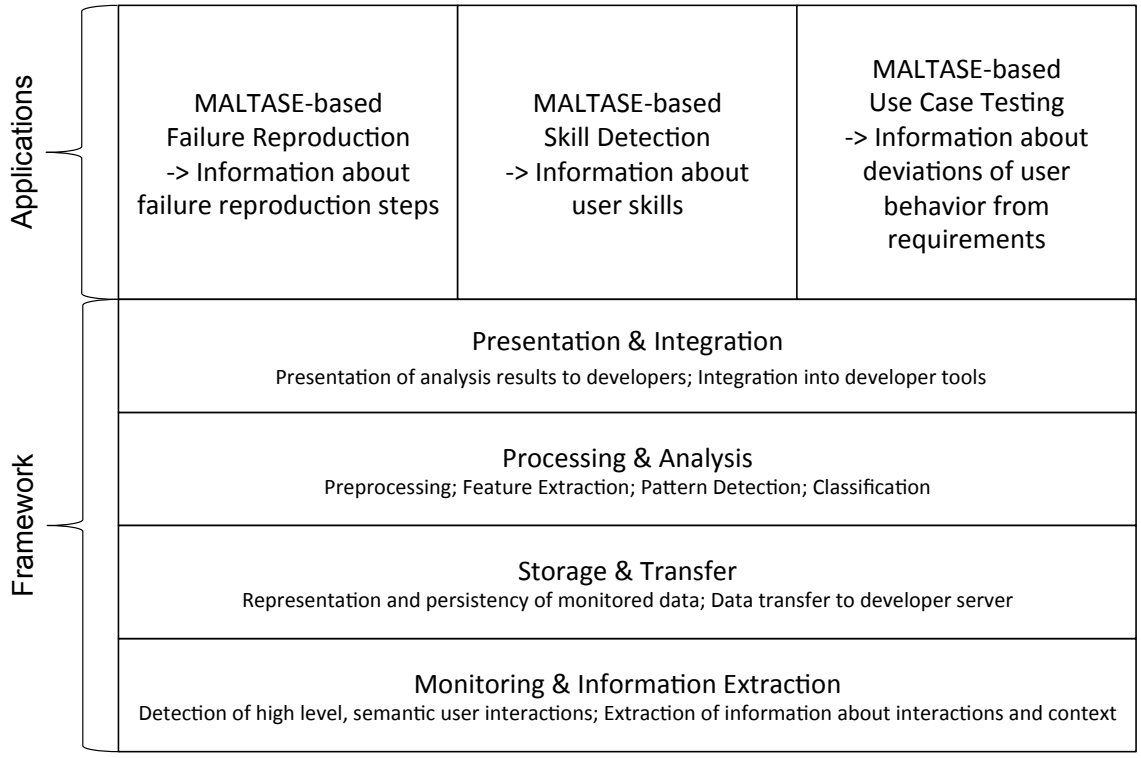


Figure 4.1: Overview of MALTASE Framework And Framework Applications

MALTASE can be employed for different purposes. We call these purposes "framework applications". Figure 4.1 shows three framework applications: MALTASE-based failure reproduction denotes reporting monitored user interactions preceding failures to developers. We hypothesize that this information supports developers when reproducing failures and fixing bugs as reproduction steps are often missing in bug reports submitted by users. MALTASE-based skill detection denotes inferring user skill levels from monitored user interactions. We hypothesize that this information provides developers with insights about their user population and enables a target application to adapt to the current user. MALTASE-based use case testing denotes comparing monitored user interactions to the flow of events of the use case specification. As the use case specification represents developer assumptions about user behavior, we hypothesize that MALTASE-based use case testing detects mismatches between developer assumptions and user behavior which may lead software improvements or use case updates. These framework applications are described in more detail in Chapter 5 and an empirical evaluation of MALTASE-based failure reproduction and MALTASE-based skill detection is presented in Chapter 6.

4.2 Requirements

This section presents requirements for the MALTASE framework by describing functional requirements, non-functional requirements, and use cases.

The overall goal of MALTASE is to collect usage information and extract helpful knowledge for developers during software evolution. Thereby, it aims to close the

problems of missing developer knowledge about usage of the target application and communication gaps between developers and users during software evolution.

4.2.1 Functional Requirements

To achieve its overall goal, the MALTASE framework should implement the functional requirements described below.

Monitoring of User Interactions With Semantics and Context User interactions should be monitored at a high level of abstraction to be interpretable by developers and enable developers to understand into user behavior. For example, the user interaction "Save File" is more semantic as the user interaction "Mouse click on screen position [100, 30]", while both represent the same user interaction of saving a file by clicking on a toolbar icon. Therefore, user interactions should be monitored abstraction levels as high as possible such as "UI Events" and "Abstract Interaction" (see Hilbert and Redmiles [79]). Furthermore, the context of a user interaction should be captured. The appropriate level of abstraction as well as the definition of context depends on the analysis purpose. Hence, the instrumentation and analysis components have to be extensible and allow developers to adjust them according to their current information need.
To enable the evaluation of MALTASE-based failure reproduction and MALTASE-based skill detection, the MALTASE framework should implement sensors which collect the information necessary for both framework applications. They are described in more detail in Chapter 5.

Temporal Storage of Monitored Data on User Device Monitored data should be stored temporarily on the user device until its transfer to a developer server.

Transfer of Monitored Data to Developer Server Before monitored data can be analyzed and inspected, it has to be transferred to a server on the developer's site. It should be able to trigger this data transfer by users manually, by the occurrence of special events (such as closing the target application), or continuously during the interaction of users with a target application.

Storage of Monitored Data on Developer Server Monitored data should be stored on a developer server after its transfer. Analysis components then process and analyze this data to extract information.

Data Analysis Before data analysis can be performed, monitored data usually has to be cleaned and mining features have to be extracted. Hence, preprocessing operations such as removing undesired types of interactions should be implemented as part of an analysis component. Similarly, feature extraction operations such as extracting the occurrence frequency of different types of user interactions should be implemented. These preprocessing and feature extraction operations can be reused and extended by developers designing a certain type of data analysis.

The purpose of data analysis is to extract helpful information from monitored data, i.e. insights about the user, the target application, or the interaction between both. Depending on the type of information desired, the data analysis varies. Hence, the MALTASE framework should implement different types of analysis algorithms such as classification learning or frequent pattern mining. These analysis algorithms can be reused or extended by developers designing a data analysis routine for a particular information need.

To enable the evaluation of MALTASE-based failure reproduction and MALTASE-based skill detection, the MALTASE framework should implement preprocessing operations, feature extraction operations, and analysis algorithms which are necessary for both framework applications. They are described in more detail in Chapter 5.

Visualization and Presentation of Monitored Data and Analysis Results Finally, monitored data and analysis results have to be presented to developers to enable them to exploit the knowledge gained. Hence, functionality for the presentation and visualization of monitored data and analysis results should be implemented by the MALTASE framework. Furthermore, it should be possible to integrate the components implementing this functionality in CASE tools used by developers and therefore in the developer's workflow.

To enable the evaluation of MALTASE-based failure reproduction and MALTASE-based skill detection, the MALTASE framework should implement functionality to present and visualize information which is necessary for both framework applications. They are described in more detail in Chapter 5.

4.2.2 Use Cases

Figure 4.2 provides an overview of the use case model of MALTASE. Two types of actors considered: users and developers of a software application.

Work With Instrumented Application The actor *user* has only this use case. In this use case, a user interacts with an instrumented target application and his or her interactions are monitored in the background. The user should not notice the presence of the instrumentation in terms of performance overhead, i.e. the target application should behave the same way as without the instrumentation.
The user should be made aware of the instrumentation and be told what type of data is collected and what it is used for.

In contrast to the actor *user*, the actor *developer* has several use cases which are described below. The use cases illustrate how the MALTASE framework implements a communication channel between users and developers via the monitored user interactions. As the use cases further show, this communication channel does not require any effort by users, besides agreeing to use an instrumented target application. In contrast, the actor *developer* has to perform some effort to establish this communication channel and exploit information gathered. At least, developers have to instrument an existing target application and deploy it to one or more users to collect data. Furthermore, they have to conduct an analysis of monitored data to gain usage information.

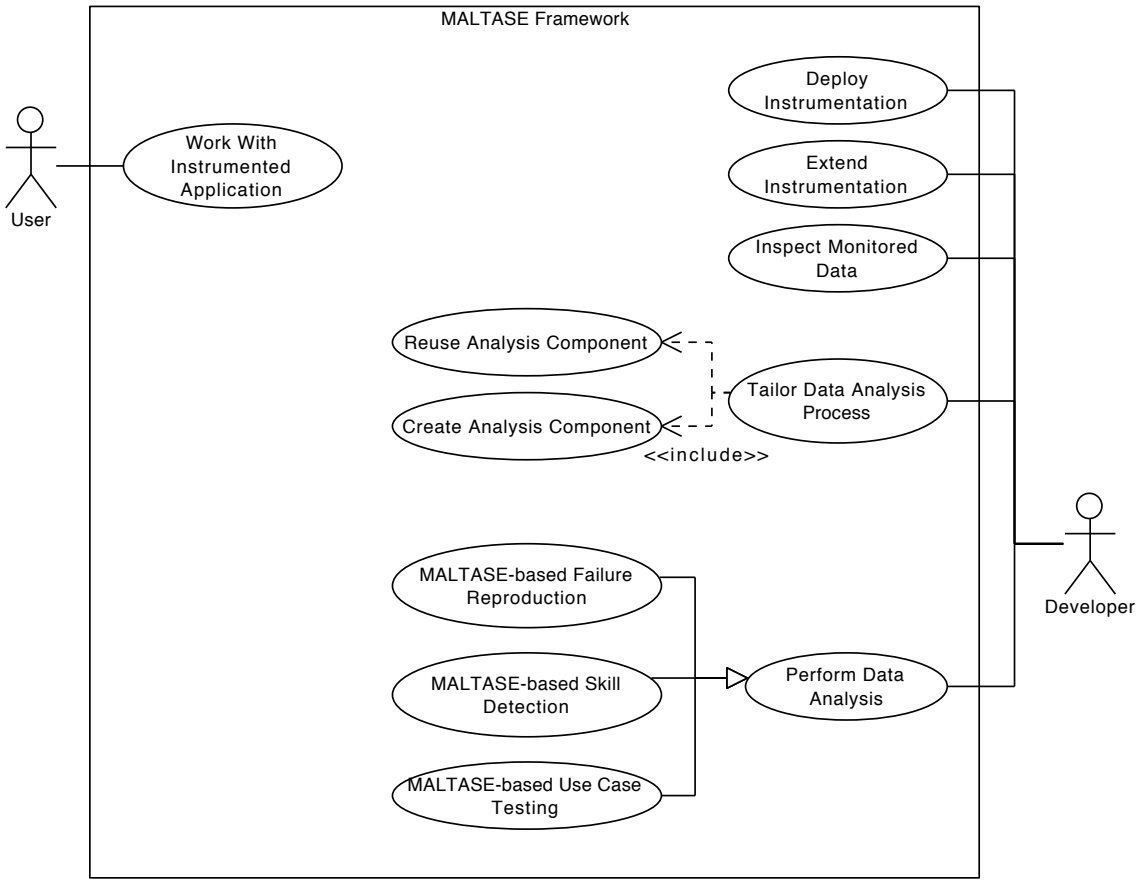


Figure 4.2: Use Cases of MALTASE Framework

Deploy Instrumentation The developer has to instrument an existing target application and deploy the instrumented target application to a user. To accomplish this task, the developer uses sensors as well as storage and transfer infrastructure of the MALTASE framework.

Extend Instrumentation The developer extends sensors of the MALTASE framework toto monitor additional details about user interactions or interaction context. Thereby, the developer reuses the existing storage and transfer infrastructure of the MALTASE framework.

Inspect Monitored Data The MALTASE transfer component transfers monitored data from user devices to a developer server. The developer inspects this data to check which data arrived at the server and manually analyze it.

Tailor Data Analysis Process The developer creates a new analysis by reusing and combining existing analysis components (use case "Reuse Analysis Component"), implementing new analysis components (use case "Create Analysis Component"), or a mixture of both.

Reuse Analysis Component The developer reuses an existing analysis component such as a classification algorithm. This use case requires that a data analysis routine for the current information need is not yet implemented but the required analysis algorithm is already implemented as part of the MALTASE framework.

Create Analysis Component The developer creates a new analysis component which analyses monitored data in a new, not yet implemented way.

Perform Data Analysis The developer analyses monitored data to extract information satisfying his or her current information need. This is the central use case for the actor developer as it aims to provide him or her with helpful knowledge, while the other use cases were preparing this use case. Each framework application of the MALTASE framework constitutes a specialization of this use case.

Maltase-based Failure Reproduction The developer retrieves monitored user interactions preceding a failure from the datastore and exploits them as reproduction steps. This use case is a specialization of the use case "Perform data analysis" and corresponds to the framework application "MALTASE-based failure reproduction" which is described in Section 5.1.

Maltase-based Skill Detection The developer analyzes monitored user interactions to infer user skill levels automatically. Skill information supports the developer in his or her evolution decisions. This use case is a specialization of the use case "Perform data analysis" and corresponds to the framework application "MALTASE-based skill detection" which is described in Section 5.2.

Maltase-based Use Case Testing The developer compares monitored user interactions to the flow of events from the use case specification automatically. Detected mismatches between both indicate software improvements or use case updates. This use case is a specialization of the use case "Perform data analysis" and corresponds to the framework application "MALTASE-based use case testing" which is described in Section 5.3.

4.2.3 Non-Functional Requirements

To achieve its overall goal, the MALTASE framework should support the non-functional requirements described below.

Limited Performance Overhead Every instrumentation introduces a performance overhead compared to a plain version of the target application. To enable use in real life situations, the MALTASE sensors should introduce a performance overhead which does not hinder users in their work with the target application.

Maintaining User Privacy Privacy questions arise when monitoring user interactions with a target application and users might not accept MALTASE because of privacy issues. Hence, monitored data should not allow to identify the current user.

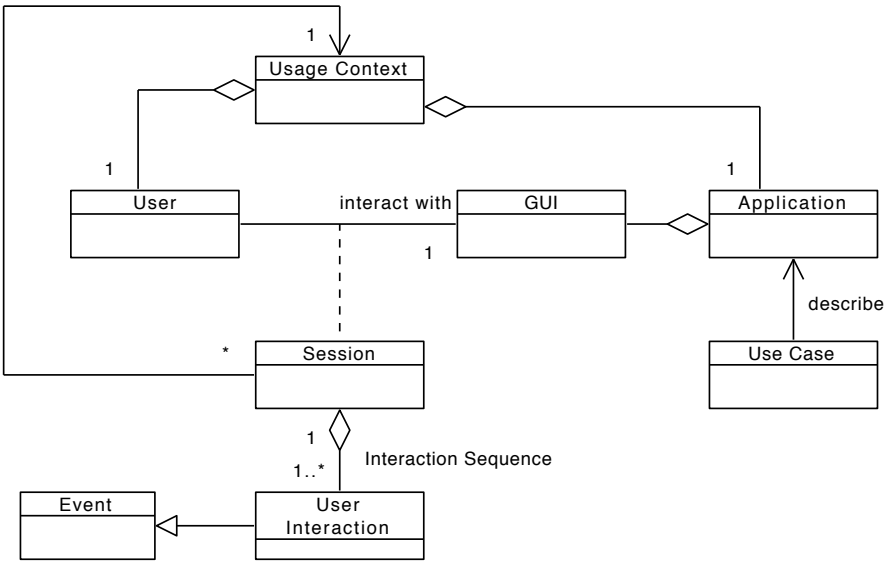


Figure 4.3: Overview of MALTASE Model

Feasible Integration With Target Application It should be possible to integrate the MALTASE framework with a target application with low effort. Obviously, the implementation technology (programming language, APIs) of the target application have to be considered here. When the target application uses the same implementation technology as MALTASE , it should be possible to integrate MALTASE framework and target application easily. When the target application uses another implementation technology as MALTASE , it should be possible to reuse the components for storage, data analysis, and presentation; only the monitoring component should have to be adapted.

Extensibility As the type of data analysis depends on the current developer information need, the components for instrumentation and data analysis should be extensible. The extensibility should allow developers to reuse and extend existing components to tailor their own monitoring and data analysis procedure.

4.3 MALTASE Model

The most important abstractions in the MALTASE model are users, applications, the interaction between both, as well as the usage context in which interactions occur. Figure 4.3 shows how these concepts are related. An individual *User* works with a particular *Application*. The interaction between the *User* and the *Application* via the *GUI* is divided into sessions which consists of a sequence of *User Interactions*. Each *Session* occurs in a certain *Usage Context* which consists of the *User,* the *Application,* and further information depending on the type of analysis. A *Use Case* describes a certain functionality of the target application.

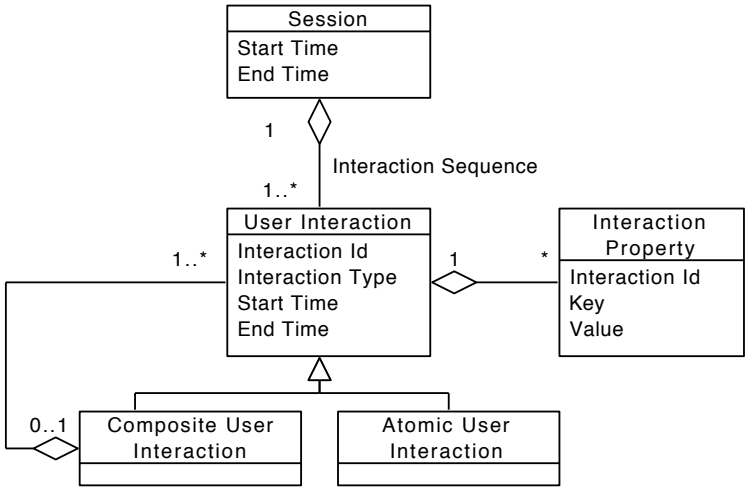


Figure 4.4: Model of User Interactions

4.3.1 Model of User Interactions

The model depicted in Figure 4.4 extends the model of Figure 4.3 and provides additional details how user interactions are modeled. Each *User Interaction* has an attribute Interaction Id which uniquely identifies this particular interaction instance, an attribute Interaction Type representing the type of the interaction within a taxonomy of user interactions, and attributes Start Time and an End Time. Further, each User Interaction may have an arbitrary number of *Interaction Properties,* allowing to capture additional information specific to a particular interaction type. As already discussed in the background section 2.2.4, user interactions can be captured at different levels of abstraction and interactions on an abstract level usually consist of several interactions on a lower level of abstraction. To reflect this structure and to enable the modeling of user interactions on different levels of abstraction, the composite pattern (cf. Gamma et al. [58]) is used: The class *Atomic User Interaction* represents a single user interaction and the class *Composite User Interaction* represents a sequence of user interactions, possibly from a lower level of abstraction.

As discussed in background section 2.2.4, interactions between users and a GUI can be considered on different levels of abstraction. Figure 4.5 shows the abstraction levels considered by the MALTASE model. It is based on the abstraction levels defined by Hilbert and Redmiles [79].

The following example introduces different abstraction levels before the abstraction levels of user interactions in MALTASE are discussed in more detail. The example considers the situation where a user works with a presentation software with the goal to create a slide deck for a presentation. The user clicks on the "new slide" button to create a new, empty slide and add it to the slide deck. The action of the user to press the button on the mouse with his or her fingers is a *Physical Event*. The device controller issues an interrupt to inform the system that the mouse was clicked, which constitutes an *Input Device Event*. On the next level, *UI Event*, the particular button which was clicked is known, namely "new slide". Clicking this button represents the *Abstract Interaction* "adding a new slide" which is part of the

Task to design a slide. Designing slides constitutes a sub task of the overall *Goal* to create a slide deck for a talk.

As a user interaction on a particular abstraction level is usually composed of several user interactions on a lower level of abstraction, the frequency of user interactions decreases with increasing abstraction level. This has direct implications for the performance overhead introduced by an instrumentation as each monitored user interaction event has to be processed and stored. Similarly, the semantics of a user interaction increases with abstraction, e.g. the user interaction "mouse click at position [237, 25]" (input device event) has less semantic information as the interaction "Save file" (abstract interaction).

As further shown by Figure 4.5, MALTASE considers three different types of user interactions: *Artifact Manipulation*, *Command*, and *GUI Interaction*. Users interact with a software application not as a goal in itself but to get their work done. MALTASE models the work users are performing when employing a target application as the manipulation of digital artifacts. Examples of digital artifacts are presentations, slides, source code, or UML models. Users manipulate those artifacts by operations like creation, deletion, or modification. The MALTASE model considers such manipulations of digital artifacts and labels them *Artifact Manipulation*. As artifact manipulations are more abstract than interactions with GUI elements and potentially consist of several such interactions, they constitute interactions on abstract interaction level. Therefore they provide semantics to user behavior.

The MALTASE model considers a second type of user interaction on abstract interaction level, namely *Commands* such as opening a file, closing the target application, or copy and paste interactions. Usually these commands can be triggered in several ways like hot keys, the icon bar, or the main menu. Hence, they are more abstract than interactions with GUI elements and provide semantics for user behavior.

Finally, the MALTASE model considers interactions of users with GUI elements which are labeled *GUI Interaction*. Each GUI interaction considers one or more elements of the GUI such as buttons or windows. Types of GUI interactions can be the activation of a GUI element or providing user input to a GUI element. As these interactions are targeted to specific GUI elements, they are on UiEvent level.

The taxonomy of user interactions depicted in Figure 4.6 models the types of user interactions considered in the MALTASE model in more detail and puts them in the context of events.

4.3.2 Model of Users, Events, Applications, and Usage Contexts

Model of Users

The class *User* represents users which can be identified by their attribute *Anonym Name,* a user name that allows to determine whether two sessions originate from the same user but does not allow to identify the user person.
As users can be distinguished with regard to their usage purpose and their background, different characteristics of an individual user are represented by the class *User Property*. Examples of user properties are their age or their skill. User-related aspects such as the current task, the current location, or the current social situation

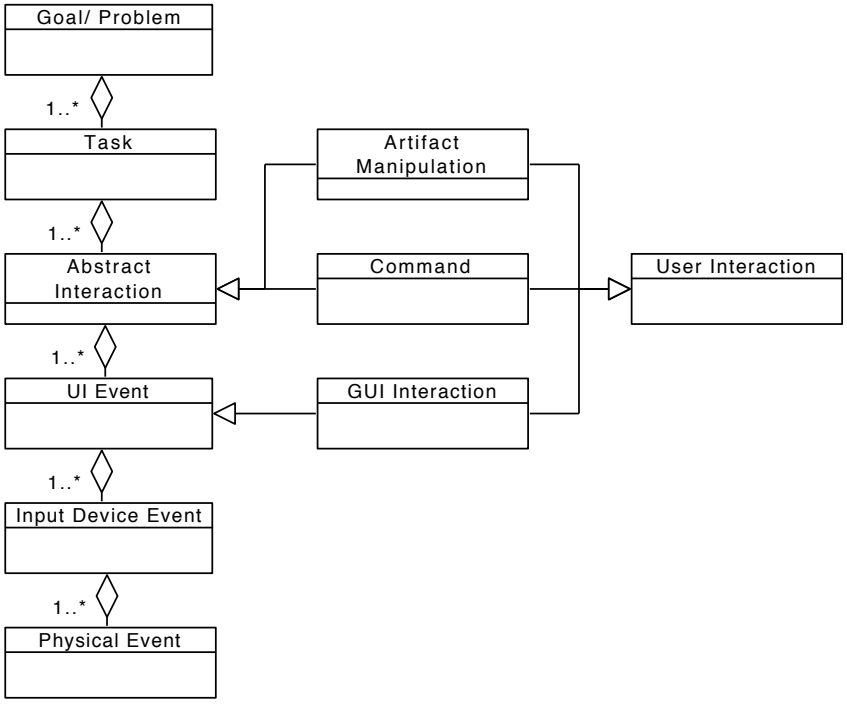


Figure 4.5: User Interactions on Different Levels of Abstraction (Adapted from [79])

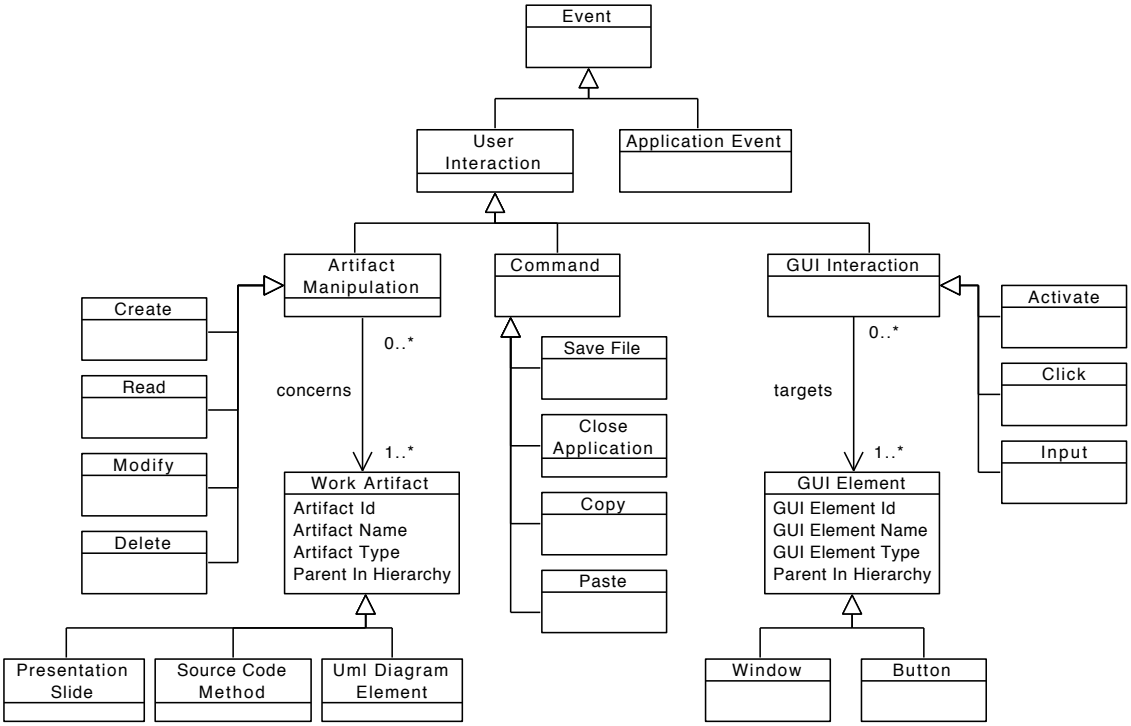


Figure 4.6: Taxonomy of User Interactions

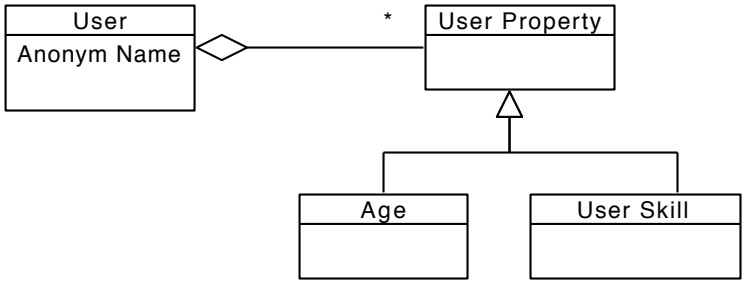


Figure 4.7: Model of Users

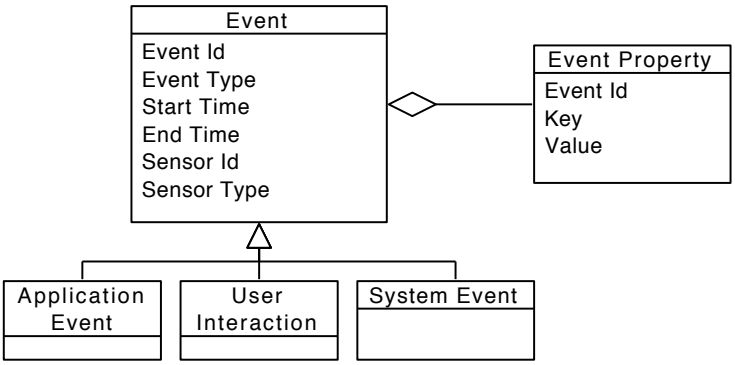


Figure 4.8: Model of Events

are modeled as part of the usage context which is described below. Figure 4.7 details the general model from Figure 4.3 and shows these relationships. The model of users can be easily extended if other user characteristics should be considered.

Model of Events

Figure 4.8 shows how events are represented in the MALTASE model. Events are the basic mechanism to represent monitored data. Each *event* has the attributes Event Id, an Event Type, a Start Time and End Timestamp, and a reference to the sensor which created the event, represented by the attributes Sensor Id and Sensor Type. Figure 4.8 shows the first level of the event taxonomy with three types of events: User Interactions (actions of users, see Figure 4.4 and Figure 4.6 for details), Application Events (events within the target application during execution, see Figure 4.10 for details), and System Events (events within the system of the target application during execution). The attribute Event Type characterizes the event. It can refer to a particular ontology class in the event ontology. Additionally, every event can have an arbitrary number of key-value properties which are modeled by class *Event Property*. Event properties capture additional information about an event and allow to capture information which is specific for certain types of events.

Model of Applications

Figure 4.9 provides an overview of the model of software applications. As we are studying the interaction between users and software applications, the most relevant

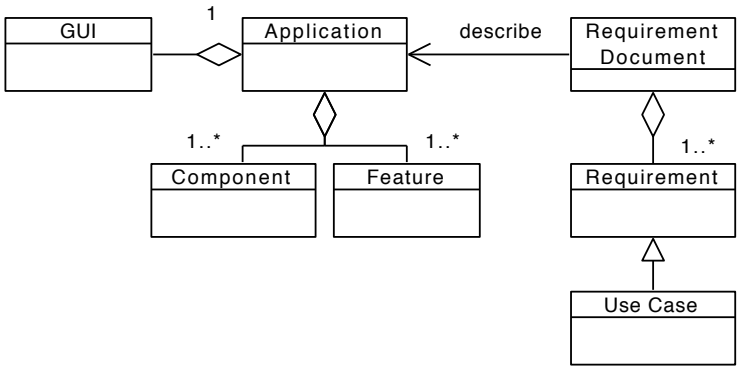


Figure 4.9: Model of Applications

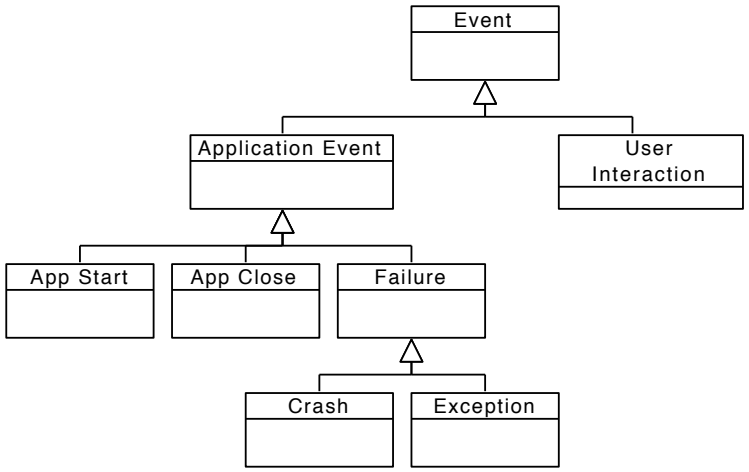


Figure 4.10: Taxonomy of Application Events

part of the target application is the user interface. We assume that the target application has a graphical user interface. Other aspects of the target application are its components and its features. The components are the libraries or software modules which collaboratively constitute the target application. The features are the blocks of functionality which the target application offers to the user. The requirements document consists of a set of requirements which describe the functionality of the target application. An example for a requirement is a use case. This model of applications can be easily extended if other application aspects should be considered.

During the execution of a target application, different types of events occur. Similarly to user interactions, those events can be monitored and logged. Which types of events are relevant depends on the data analysis intended by developers. Figure 4.10 shows a taxonomy of application events. It contains events representing starting the target application (Class App Start), closing the target application (Class App Close) as well as two kinds of failures: Crash and Exception. This taxonomy can be easily extended if other application events should be considered.

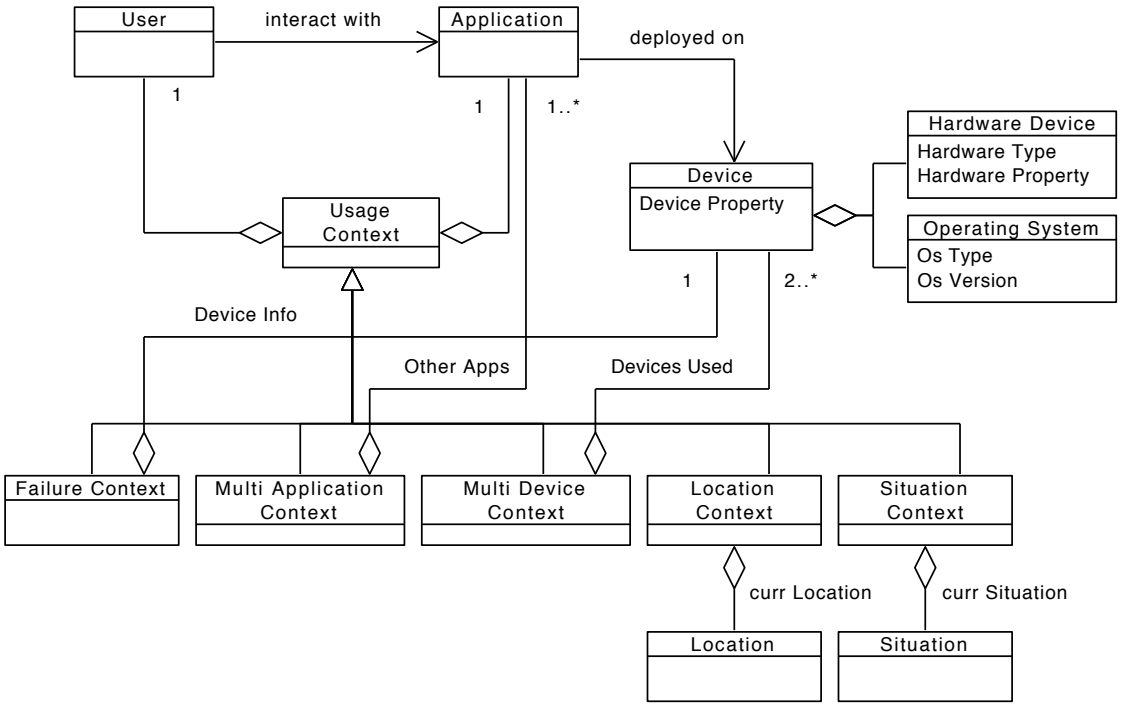


Figure 4.11: Model of Usage Contexts

Model of Usage Context

The Class *Usage Context* represents information about the context in which a user interacts with a target application. Hence, each usage context contains information about the current user and the current target application. Which context information is relevant beyond this basic information depends on the type of analysis. Extended usage contexts are shown in Figure 4.11: *Failure Context* represents the context in which a failure occurs. To reproduce the failure and fix the corresponding bug, information about user interactions preceding the failure (captured as sequence of user interactions) as well as the current system configuration (represented as Device Info in the model) are necessary. *Multi Application Context* represents a user working with several software applications in parallel on the same device and switching between them. *Multi Device Context* represents a user using the same target application on different devices, e.g. on a laptop, a tablet, and a smartphone. *Location Context* represents a context in which the current location of the user is relevant. Finally, *Situation Context* considers the current situation of the user, e.g. whether the user currently drives a car, participates in a meeting, or sits at his or her desk. Other types of usage context can be added to extend the model.

4.4 MALTASE Architecture

The MALTASE framework is designed using a layered architecture style [25] and consists of four layers: Monitoring and Information Extraction layer, Storage and Transfer layer, Processing and Analysis layer, and Presentation and Integration

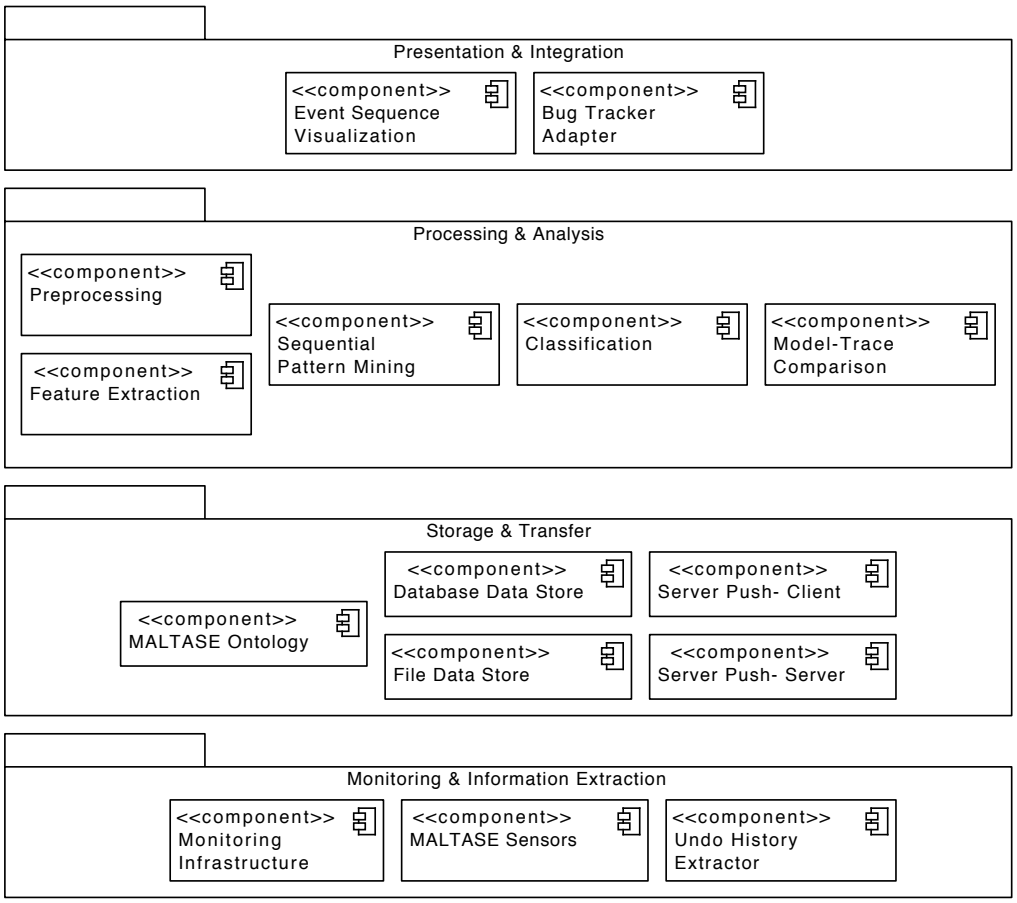


Figure 4.12: Overview of Framework Layers and Components

layer. Each layer implements a subset of the framework functionality. Figure 4.12 depicts the layers and their main components.

Monitoring & Information Extraction Layer The components of this layer implement functionality to detect user interactions and extract context information. An important aspect is the integration with the target application. The integration approach used depends on the implementation technology of the target application as well as the instrumentation approach chosen. This layer provides two approaches to monitor user interactions: The component MALTASE *Sensors* implements sensors which register as event listeners in the target application or GUI framework. While a user interacts with the target application, events are generated by the application framework, caught by the sensors, and user interactions are identified. Because the sensors are implemented as event listeners, they depend on the framework used, e.g. Eclipse RCP or SWT, but not on the particular target application. Each sensor targets a certain type of user interactions. Furthermore, the component *Undo History Extractor* provides another approach to monitor user interactions. In contrast to the user interaction sensors, user interactions are not monitored when they occur. Instead, the interaction history of the undo feature, i.e. the history of user actions which is maintained by many software applications to allow users to undo previous

actions, is extracted. The extraction can be triggered by special events such as the occurrence of a failure or can be done periodically. Obviously, this monitoring approach is applicable only for software applications supporting an undo feature. But for those software applications, it constitutes a monitoring approach without performance overhead as the manipulation actions are captured anyway. The component *Monitoring Infrastructure* implements functionality to represent interaction data, control sensors, and access the data store. This component is independent of the application framework and target application, but depends on the implementation technology.

Storage & Transfer Layer The components of this layer implement functionality to store monitored data and transfer it to a developer server for further processing and analysis. The layer implements two types of data store, namely a *File Data Store* which saves monitored data as XML files and a *Database Data Store* which saves monitored data in a relational database. In both cases, the MALTASE *Ontology* of application events and user interactions can be used as data model when persisting monitored data. This eases data integration from different sensors and enables ontology reasoning on monitored data. Using ontologies is optional and sensor implementors can decide to use self-defined event names and property names. There are two ways how monitored data can be transferred to a developer server. The user can manually send one or more files with monitored data to developers (not depicted in Figure 4.12). Also, monitored data can be sent automatically from the user device to the developer server by the server push mechanism. The server push mechanism is implemented by the components *Server Push-Client*, which runs on the user device, and *Server Push-Server*, which runs on the developer server. The data transfer can be trigger by specific events such as closing the target application, periodically, or continuously during application usage.

Processing & Analysis Layer The components of this layer implement functionality to process monitored data and extract relevant information for developers. There are two types of components within this layer: components to preprocess data and extract features as well as components implementing machine learning algorithms. The component *Preprocessing* subsumes functionality to preprocess and clean monitored data. It implements functionality such as removing certain types of interactions. The data for preprocessing operations are retrieved from the data store. After data preprocessing, the component *Feature Extraction* may extract attributes that form the input for machine learning algorithms. This is the case when it is not reasonable to use monitored, cleaned data directly. Examples of features are the average duration of breaks between two user interactions or the number of interactions per minute. After preprocessing and feature extraction, different machine learning algorithms can be applied to extract patterns and knowledge. Different types of analysis are represented by the components *Sequential Pattern Mining* (mining sequential patterns that occur frequently in interaction traces), *Classification* (classifying a user interaction sequence or a user into a set of pre-defined classes), and *Model-Trace Comparison* (Comparison of interaction traces to a model of expected behavior).

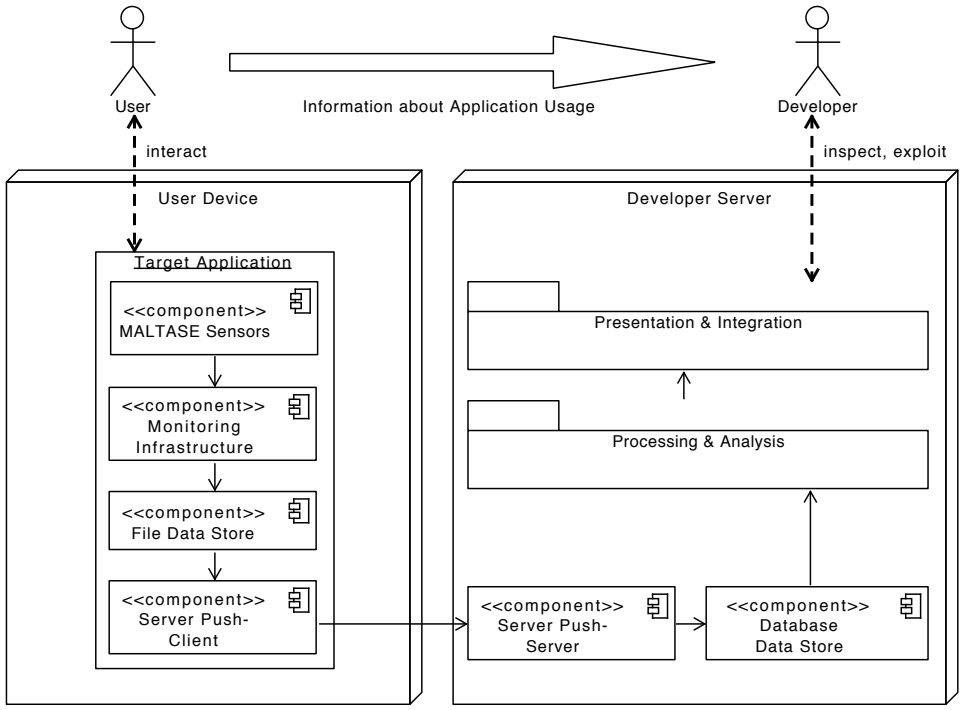


Figure 4.13: Deployment of Framework Components
Arrows denote data flow between components

Presentation & Integration Layer The components in this layer implement functionality to present monitored data and The component *Event Sequence Visualization* implements functionality to visualize a sequence of monitored user interactions and their context. The component *Bug Tracker Adapter* implements functionality to create bug reports automatically and add monitored data or analysis results.

Deployment of Components Figure 4.13 shows a deployment diagram of the MALTASE components. Components are deployed on the user device and the developer server. Figure 4.13 shows only one user device but there might be many user devices communicating with the same developer server. The user interacts with the target application running on the user device. The target application is instrumented with the MALTASE Sensors. Monitored data is stored temporarily in a File Data Store on the user device. From this temporal data store monitored data is transferred to the developer server by the server push mechanism which is implemented by the components Server Push-Client, which is deployed on the user device, and Server Push-Server, which is deployed on the developer server. The Server Push-Server receives monitored data sent by the Server Push-Client and stores it in a Database Data Store on the developer server. All components pf the preprocessing and analysis layer and the presentation and integration layer operate on this data.

Overall, this architecture enables the flow of information about software usage from users to developers. Developers get information about the interactions of users and can analyze them regarding different aspects of interest. Thereby, MALTASE

addresses the problem of missing developer knowledge about software usage as well as communication gaps between developers and users.

4.4.1 Monitoring & Information Extraction Layer

The purpose of the Monitoring and Information Extraction layer is to detect user interactions when they occur and to extract information about user interactions and their context. This functionality can be implemented in different ways depending on the targeted abstraction level of user interactions, the implementation technology of the target application, and the integration mechanism into the target application.

MALTASE uses an instrumentation approach which instruments the application framework of target applications. Software applications and their GUIs are rarely developed from scratch but using existing application and GUI toolkits. An example of an application toolkit is the Eclipse RCP framework while the Standard Widget Toolkit (SWT), Swing, and Abstract Window Toolkit (AWT) are examples for GUI toolkits. Instrumenting software applications on this level has the advantage that an instrumentation can be reused for every target application which is developed with the same framework.
The MALTASE instrumentation consists of two parts, the Monitoring Infrastructure Component and a set of sensors. The *Monitoring Infrastructure* component implements functionality to represent monitored data, to communicate with a datastore, and to manage a set of sensors. This component can be seen as a library implemented in a particular programming language and it depends only on the programming language used. Different types of sensors detect user interactions and collect information about them and their context. The implementation of the sensors reuse the monitoring infrastructure component. They integrate the instrumentation with application or GUI toolkits and therefore depend on the toolkit used.
If another target application with the same programming language and the same toolkits is to be instrumented, both components can be reused directly. If another target application with the same programming language but other toolkits is to be instrumented, the monitoring infrastructure component can be reused and only the sensors have to be re-implemented. If another target application with different programming language and different toolkits is to be instrumented, both components have to be re-implemented.

Monitoring Infrastructure

The *Monitoring Infrastructure* component implements functionality to represent monitored data, to control a set of sensors, and to communicate with a data store. It is implemented in the Java programming language.

Monitored data are represented as events as described in the MALTASE model above (see Figure 4.8). Events are generated by sensors. A sensor is a component whose task is the detection of one particular type of event and elicit information about it. The use of sensors components constitutes a modular design as a sensor implementor has to consider only how to detect the event type of interest and sensor implementations are independent of each other. Figure 4.14 gives an overview of the monitoring infrastructure component. The class *Sensor Manager* controls the

whole instrumentation which is composed of one or more sensors. The whole instrumentation can be started, paused, or terminated via the sensor manager. Pausing the instrumentation denotes stopping it temporarily and it can be resumed later. Terminating the instrumentation denotes stopping the instrumentation completely and it cannot be restarted. The sensor manager is implemented as a singleton (see Gamma et al. [58]) to ensure that there is only one instance of it. When a *Sensor* is created, it has to register itself at the sensor manager. Similar to the sensor manager, each sensor implements operations to start, pause, and terminate the sensor. Important operations for the sensor are initialize and deinitialize which have to be implemented by every implementation of the abstract sensor class. The initialize operation sets up the sensor by registering it as listener for specific framework events or starting a new monitoring thread. The deinitialize operation is its counterpart and cleans up everything the initialize method set up. The sensor class is abstract and implements general functionality of a sensor. The specific functionality how the occurrence of an event is detected and how further information is elicited has to be implemented by subclassing the sensor class and overwriting the initalize and deinitialize methods.

When a sensor detects an event, it creates an instance of the class event which represents the detected event and captures event information in the attributes and properties of this instance. It does not call specific components for further processing or storage of the event, but announces the event by publishing it on the *Event Bus*. This design follows the observer pattern (see Gamma et al. [58]) and decouples sensors from further processing and storage components. It allows easy addition and removal of sensors and event listeners. The event bus represents the subject class of the observer pattern and notifies all registered listeners about the occurrence of an event. It is also implemented as a singleton (see Gamma et al. [58]) to ensure that there is one instance. The *Event Listener* class represents the observer class of the observer pattern. Each implementation has to overwrite the handle event method which is called when an event is published on the event bus. An event listener has to be registered at the event bus to be notified of published events. Examples for event listeners are the classes *File Writer* and *Database Adapter* that implement functionality to write the event sequences to a file or a database.

MALTASE Sensors

This component constitutes of a set of sensors which monitor user interactions and application events of a desktop application. Table 4.1 provides an overview of the sensors. The sensors instrument software applications implemented using the Eclipse Rich Client Platform RCP [1] and the Eclipse Standard Widget Toolkit SWT[2]. The sensors integrate with these frameworks by using well defined extension points. Hence, they can be installed and removed using the Eclipse update mechanism. Furthermore, the target application does not have to be modified or recompiled to install the sensors.

[1] http://wiki.eclipse.org/Rich_Client_Platform (Accessed Jan 2015)
[2] http://www.eclipse.org/swt/ (Accessed Jan 2015)

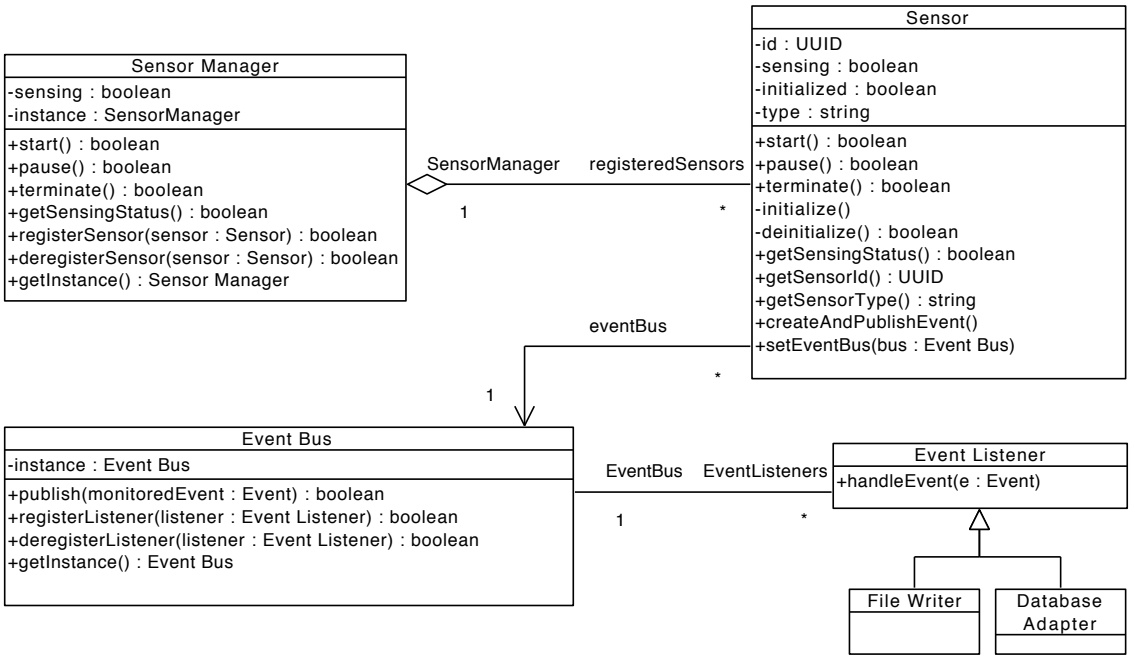


Figure 4.14: Object Design of Component Monitoring Infrastructure

Table 4.1: Overview of MALTASE Sensors

Sensor Name	Monitored Events	Abstraction Level	Dependency of Implementation
Sensors for User Interaction Events			
Command Sensor	User commands such as "save file"	Command (Abstract interaction)	Application framework (Eclipse RCP, SWT)
Menu and Toolbar Sensor	Interactions of users with main menu, context menus, and tool bar	GUI interaction (UI Event)	Application framework (Eclipse RCP, SWT)
GUI Part Sensor	Activation of GUI parts such as windows	GUI interaction (UI Event)	Application framework (Eclipse RCP)
Diagram Manipulation Sensor	Diagram manipulations such as adding an UML class	Artifact manipulation (Abstract interaction)	Modeling Framework (EMF)
Sensors for Application Events			
Application Sensor	Starting and closing of target application	Application Event	Application framework (Eclipse RCP)
Exception Sensor	Unhandled exceptions	Application Event	Application framework (Eclipse RCP)

User Interaction Sensors

The following user interaction sensors have been implemented:

- Command Sensor
  This sensor monitors user commands such as saving files or exporting documents. Artifact manipulations are also covered by this sensor if they are implemented using the command mechanism. It hooks as listener into the SWT event queue and the Eclipse RCP command queue.

- Menu and Toolbar Sensor
  This sensor monitors interactions with the main menu, context menus, and the tool bar. It hooks as listener into the SWT event queue and the Eclipse RCP command queue.

- GUI Part Sensor
  This sensor monitors the activation of GUI parts such as windows or dialogs. It hooks as listener into the Eclipse workbench infrastructure.

- Diagram Manipulation Sensor
  This sensor monitors manipulations of diagrams such as a UML diagrams. It hooks as listener into the Eclipse EMF framework[3], which is a Eclipse framework for creating and manipulation diagrams.

Figure 4.15 exemplary shows how the Command sensor integrates with the Eclipse RCP framework. It extends the abstract sensor class from component Monitoring Infrastructure and implements the initialize and deinitialize methods. In the initialize method, it registers itself as listener for SWT selection events at the display of the target application. Therefore, it must implement the interface Listener and overwrite its method handleEvent. This method is called when a selection event occurs and it further analyzes the selected widget. When the widget is of type Item, additional information such as the command id and the label text of the menu item are extracted. Extracted information is captured in a new instance of class Event.

Application Event Sensors

The following application event sensors have been implemented:

- Exception Sensor
  This sensor monitors handled exceptions during the execution of a target application. It hooks as listener into the Eclipse runtime platform.

- Application Sensor
  This sensor monitors when the target application starts and closes. Besides generating events which represent these events, this sensor is used to trigger start and stop the whole instrumentation. It hooks as listener in the Eclipse runtime environment.

[3]https://www.eclipse.org/modeling/emf/ (Accessed Jan 2015)

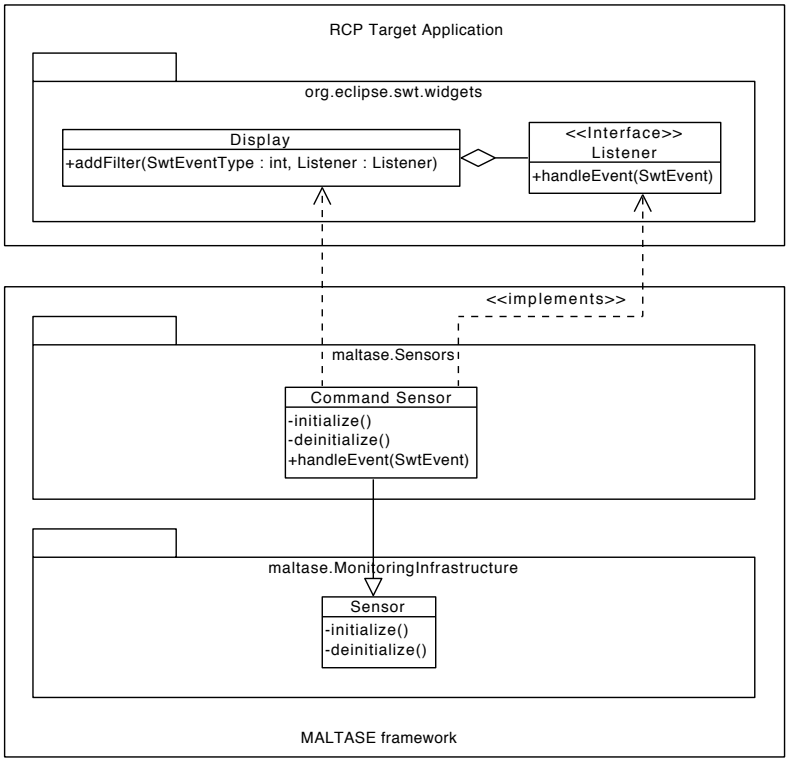


Figure 4.15: Integration of Command Sensor Into RCP Target Application

Undo History Extractor

The user interaction sensors described above integrate with application or GUI toolkits. This design ensures reusability for target applications developed using the same frameworks but introduces a performance overhead for each single event. In contrast, the component Undo History Extractor does not monitor user interactions directly but extracts them from the action history of undo features. It exploits the following observation: If a target application allows users to undo previous actions, a history of user actions is already maintained to implement the undo functionality. Many software applications allow a fine-grained undo of previous actions which requires the capture of user actions at a detailed level. For example, Figure 4.16 shows an undo history of Microsoft Excel (order of actions from bottom to top) containing detailed information about previous user actions.
The Undo History Extractor exploits this situation. Upon certain trigger events such as unhandled exceptions or timers, it extracts the action history of the undo feature. Therefore, it allows to monitor user interactions without additional performance effort as the action history of the undo feature is captured anyway. But it also has some drawbacks. Obviously, it requires a target application supporting undo functionality. Furthermore, it relies on the actions captured by the undo feature in terms of completeness and granularity. Completeness denotes the fraction of all user interactions captured while granularity denotes the level of detail on which a single user interaction is captured. It depends on the analysis purpose of developers if information contained in the action history of the undo feature is sufficient.

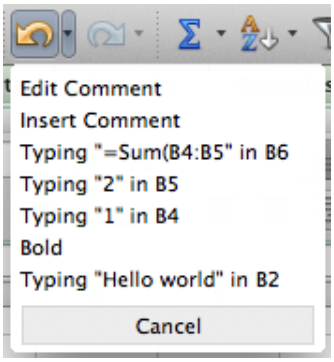


Figure 4.16: Action History of Microsoft Excel Undo Feature

The idea of the Undo History Extractor as well as an empirical study about its feasibility has been described in Roehm and Bruegge [150]. Future work should implement and evaluate it. For example, the Swing GUI toolkit provides a centralized undo manager which can be exploited to extract user actions [71].

4.4.2 Storage & Transfer Layer

The components of this layer implement functionality to store monitored data and transfer it to a developer server for permanent storage and analysis. All components are independent of the target application.

File and Database Data Store

Data monitored by MALTASE sensors has to be stored on two devices. First, it has to be stored temporarily on the user device. Then it has to be transferred to a developer server and stored there permanently. MALTASE uses two types of data store for this task: the File Data Store stores monitored data as XML files and the Database Data Store stores monitored data in a relational database. Both data stores use the event representation from the MALTASE model (see Section 4.3 for details) and translate it to an xml document structure or database schema. The Database Data Store is implemented using the MySQL database system[4]. The File Data Store is usually used on the user device. The data store on the developer server is realized with the Database Data Store because of the concurrency of multiple user devices accessing the server. Figure 4.17 shows events stored in the Database Data Store, while Figure 4.18 shows the same events represented as an xml file from the File Data Store.

Ontology Component

The MALTASE framework encompasses an ontology in the Web Ontology Language OWL [195] which represents the MALTASE model described in Section 4.3. Figure 4.19 shows an excerpt of the MALTASE ontology. It allows to define semantics

[4]http://www.mysql.com/ (Accessed Jan 2015)

	EventId	EventType	AddInfo	StartDatetime	EndDatetime
	13	ActivateGuiPart	MOSKitt UML2 Class editor	2012-12-19 16:14:35	2012-12-19 16:14:35
	14	AddUmlClass	org.eclipse.uml2.uml.Class	2012-12-19 15:56:22	2012-12-19 15:56:22
	15	ActivateGuiPart	Propiedades	2012-12-19 16:13:02	2012-12-19 16:13:02
	16	ActivateGuiPart	MOSKitt Model Explorer	2012-12-19 16:09:05	2012-12-19 16:09:05
	17	SaveFile	org.eclipse.ui.file.save	2012-12-19 15:52:09	2012-12-19 15:52:09
	18	ActivateGuiPart	Propiedades	2012-12-19 16:06:09	2012-12-19 16:06:09

Figure 4.17: Monitored Events in Database Data Store

```
<?xml version="1.0" encoding="utf-8" ?>
<EventSequence>
  <Event>
    <Id>13</Id>
    <Type>ActivateGuiPart</Type>
    <AddInfo>MOSKitt UML2 Class editor</AddInfo>
    <StartTimestamp>2012-12-19 16:14:35</StartTimestamp>
    <EndTimestamp>2012-12-19 16:14:35</EndTimestamp>
  </Event>
  <Event>
    <Id>14</Id>
    <Type>AddUmlClass</Type>
    <AddInfo>org.eclipse.uml2.uml.Class</AddInfo>
    <StartTimestamp>2012-12-19 15:56:22</StartTimestamp>
    <EndTimestamp>2012-12-19 15:56:22</EndTimestamp>
  </Event>
  <Event>
    <Id>15</Id>
    <Type>ActivateGuiPart</Type>
    <AddInfo>Propiedades</AddInfo>
    <StartTimestamp>2012-12-19 16:13:02</StartTimestamp>
    <EndTimestamp>2012-12-19 16:13:02</EndTimestamp>
  </Event>
  <Event>
    <Id>16</Id>
    <Type>ActivateGuiPart</Type>
    <AddInfo>MOSKitt Model Explorer</AddInfo>
    <StartTimestamp>2012-12-19 16:09:05</StartTimestamp>
    <EndTimestamp>2012-12-19 16:09:05</EndTimestamp>
  </Event>
  <Event>
    <Id>17</Id>
    <Type>SaveFile</Type>
    <AddInfo>org.eclipse.ui.file.save</AddInfo>
    <StartTimestamp>2012-12-19 15:52:09</StartTimestamp>
    <EndTimestamp>2012-12-19 15:52:09</EndTimestamp>
  </Event>
</EventSequence>
```

Figure 4.18: Monitored Events in File Data Store

Figure 4.19: MALTASE Ontology (Excerpt)

of monitored data such as application events and user interactions. A defined semantics facilitates the integration of monitored data from different sensors as well as the collaboration of sensor implementers and implementers of analysis routines. Furthermore, it enables semantic reasoning about monitored data. Monitored events are mapped to this event ontology by specifying the correct, full URI of an ontology class as event type. If sensor implementors use ontology URIs when creating events, they are preserved during processing and storage in the MALTASE framework and can be exploited by analysis components.
Using the MALTASE ontology is optional and depends on the sensor implementors. When sensor implementors decide not to use the MALTASE ontology, it is their responsibility to assert meaningful event types which contain the information necessary for the intended data analysis.

Server Push Mechanism

Monitored data has to be transferred from the user device where it was monitored to the developer server for analysis. This functionality is implemented by the the server push mechanism which consists of two components, Server Push-Client and Server Push-Server. The component Server Push-Client runs on the user device, retrieves monitored data from a local data store, and sends it to the Server Push-Server. The Server Push-Server runs on the developer server, receives monitored data from the Server Push-Client, and stores it in a permanent data store on the developer server. The communication between client and server part is implement using TCP sockets: monitored data is serialized, transferred via sockets in a network or over the Internet, and deserialized on the server side.

An alternative to the server push mechanism is manual, file-based transfer. Monitored data is stored in local XML files on the user device. Then, it is the responsibility of the user to transfer XML files to the developer.

4.4.3 Processing & Analysis Layer

Monitoring and storing user interactions is not a goal in itself but a necessary prerequisite to analyze software usage. The components in this layer implement functionality for processing and analyzing monitored data with the goal to extract helpful knowledge for developers during software evolution. All components are independent of the target application. Analysis results are presented to developers by components of the Presentation and Integration Layer (see Section 4.4.4).

As discussed in the framework requirements in Section 4.2, the type of analysis depends on monitored data as well as the current information needs of developers. The completeness and level of detail of monitored data determines which analyses can be performed. Furthermore, different analysis might be run on the same data when developers are interested in different information. Therefore, the components of this layer constitute a toolbox of functionality. Developers can choose from it and combine different processing and analysis components depending on the available data and their analysis purpose. In some cases, processing can be skipped altogether and monitored data is inspected by developers directly. For example, monitored user interactions preceding failures can be exploited as failure reproduction steps without processing (see MALTASE-based failure reproduction described in Section 5.1).

Preprocessing Component

Monitored data usually has to be cleaned and preprocessed before data analysis can start. The preprocessing component implements such functionality.

The central entities on which all analysis components operate are *events*, *sequences of events*, and *sequence databases*. These concepts originate from the MALTASE model. An event represents user interactions or application events. An event sequence is a set of events which represents a user session. Events of an event sequence are ordered according to their timestamps. Finally, an event sequence database consists of a set of event sequences.

Figure 4.20 depicts different types of preprocessing operations. All preprocessing operations receive a database of event sequences as input, process it, and return another database of event sequences. If only one event sequence is to be analyzed, the database consists of one event sequence. Each implementation of a preprocessing operation has to extend the abstract class *PreprocessingOperation* and implement the doOperation method. This design standardizes the interface of preprocessing operations and facilitates the concatenation of different preprocessing operations.

There are five different types of preprocessing operations in the MALTASE framework: *FilterSequenceDb*, *SessionizeSequence*, *SortEvents*, *AggregateEvents*, and *FilterEventSequence*. *FilterSequenceDb* implements filter functionality on sequence database level, i.e. adding or removing whole event sequences in a event sequence database. Its subtypes implement different filter criteria: *FilterSeqDbByUser* accepts only event sequences from certain users, *FilterSeqDbByDate* accepts only event sequences fulfilling certain date requirements, *FilterSeqDbByEventType* accepts only event sequences that contain certain types of events, and *FilterSeqDbBySeqLength* accepts only event sequences which have a certain minimal or maximal length. *SessionizeSequence* implements functionality to break up a long sequence into a set of

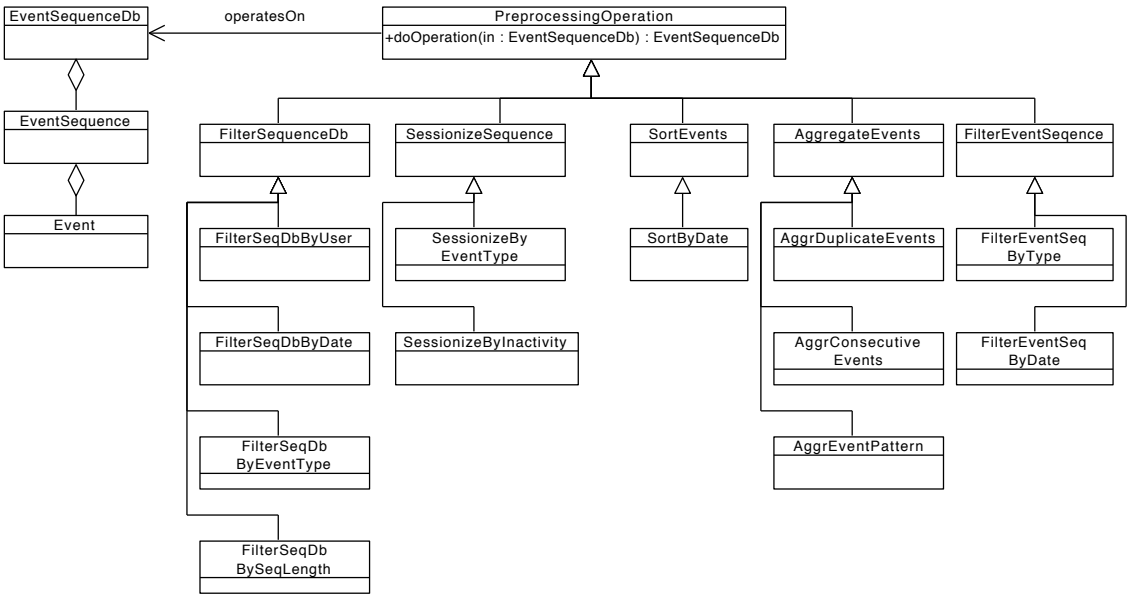


Figure 4.20: Overview of Preprocessing Operations

smaller sequences. Its subtypes implement different splitting criteria: *SessionizeByEventType* splits a sequence according to the occurrence of a certain event type, e.g. application start or application close events. *SessionizeByInactivity* splits a sequence after a certain time of user inactivity. *SortEvents* implements functionality to sort events in a sequence. For example, to sort all events according to their timestamp (*SortByDate*). *AggregateEvents* implements functionality to aggregate two or more events of an event sequence. Aggregation denotes the removal of a the set of events and replace them with a single, new event. Its subtypes implement different ways to aggregate sets of events. *AggregateDuplicateEvents* detects that two events are duplicates and removes one of them. This situation can occur when two sensors monitor the same type of event. *AggregateConsecutiveEvents* aggregates events of the same type which occur after each other without another event in between. For example, the event sequence "Copy, Paste, Paste, Paste" can be aggregated to the event sequence "Copy, Paste" if the number of Paste occurrences is not relevant. And *AggregatePattern* aggregates a certain event pattern, i.e. a set of events that occur in a specified order after each other, and replaces the pattern events with a single new event. Finally, *FilterEventSequence* implements functionality to filter events of an event sequence. Similar to the filtering on sequence level, different criteria can apply to determine which event instances to keep and which ones to remove. This functionality is used to remove noisy events which are not relevant for the analysis.

Feature Extraction Component

Many machine learning algorithms need a set of features as input. The component Feature Extraction component calculates such features from a database of event sequences. Feature extraction can be seen as an aggregation of the event sequence database focussing on aspects which are relevant for the analysis. The mechanisms in this component differ from the ones in the preprocessing component in their purpose:

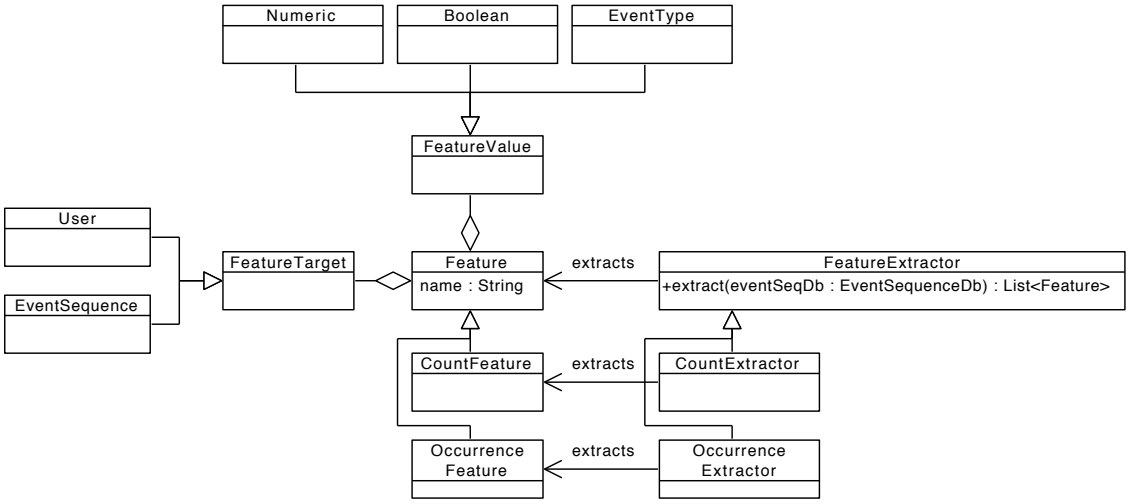


Figure 4.21: Overview of Feature Extraction Component

the purpose of the preprocessing component is to select and clean monitored data retrieved from the server data store. The mechanisms in this component represent a first step of analysis.

Figure 4.21 gives an overview of the Feature Extraction component. A *feature* consists of a *feature value* and a *feature target*. A feature can be attributed to a user or an event sequence and this fact is represented as the *FeatureTarget*. For example, the count feature calculates the number of interactions in a sequence (target: event sequence) or the number of sessions of a user (target: user). The value of a feature can be a numeric number or an event type.

Analysis is already possible at the level of extracted features. For example, event sequences can be distinguished based on whether crash events occur in them. Also, statistical analysis is possible by e. g. calculating the number of user interactions per minute or frequencies of different types of user interactions.

Sequential Pattern Mining

The purpose of this component is to mine frequent sequential patterns of events in monitored data. The component receives a database of event sequences and a support value. It mines all frequent, sequential, closed patterns which occur in the sequences of the input sequence database. Thereby, frequent denotes that the number of occurrences is higher than the support value (the minimum number of sequences in which a pattern must occur), sequential denotes that the order of events is considered, and closed denotes that there is no sub pattern with the same or higher support. This functionality is implemented using the BIDE algorithm [196] as provided by the SPMF library[5].

Detected patterns can be exploited in several ways. For example, they can be used for event aggregation: When an instance of a pattern is detected, all events of the pattern instance are removed and replaced with a single, new, more abstract event. Further they can be used as features in classification: When classifying user

[5]http://www.philippe-fournier-viger.com/spmf/index.php (Accessed Jan 2015)

skill, the occurrence of a pattern representing an undo operation can be used as an indicator for novice users.

Classification

This component implements classifiers used to classify a particular user, a particular event sequence, or a particular event into a predefined set of categories. Depending on the type of entity to be classified, the input for this component is user data, an event sequence, or a particular event. The component employs its classifier and predicts a category for the input entity.

Before the classification component can be used, a classifier has to be learned. For example, a classifier which detects wether the current user is a novice or an expert. Decision tree classifier [141] are used because decision trees are well known classifiers and decision trees can be interpreted by humans. This component is implemented using weka's J48 algorithm [69] which is an implementation of the C4.5 decision tree learning algorithm [141]. Decision tree learning is a supervised learning algorithm and therefore requires training data in order to learn the classifier.

This component is used to classify users based on their application skill in novices or experts (see MALTASE-based skill detection which is described in Section 5.2).

Model-Trace Comparison

The purpose of this component is to compare traces of monitored user interaction with a model of expected behavior and detect mismatches between both. Mismatches represent deviations of user behavior from developer expectations. We hypothesize that these mismatches yield interesting insights which can lead to software improvements or the correction of model and developer expectations.

The input of this component is a use case of the target application as well as a monitored interaction trace from a user session in which the user performs this use case. It extracts a Petri net which represents the flow of events of the use case and compares the input interaction trace to this Petri net, identifying missing interactions as well as additional interaction compared to the use case specification. The comparison is performed using the "conformance checking" mechanism of process mining [189] which is implemented in the ProM library[6].

This component is used in MALTASE-based use case detection which is described in more detail in Section 5.3.

4.4.4 Presentation & Integration Layer

The components of this layer implement functionality for presenting monitored data and analysis results to developers. To minimize the effort, they integrates into the workflow of developers whenever possible. The integration is realized by pushing information into the tools and infrastructure used by developers in their work. All components of this layer are independent of the target application.

[6] http://www.promtools.org/prom6/ (Accessed Jan 2015)

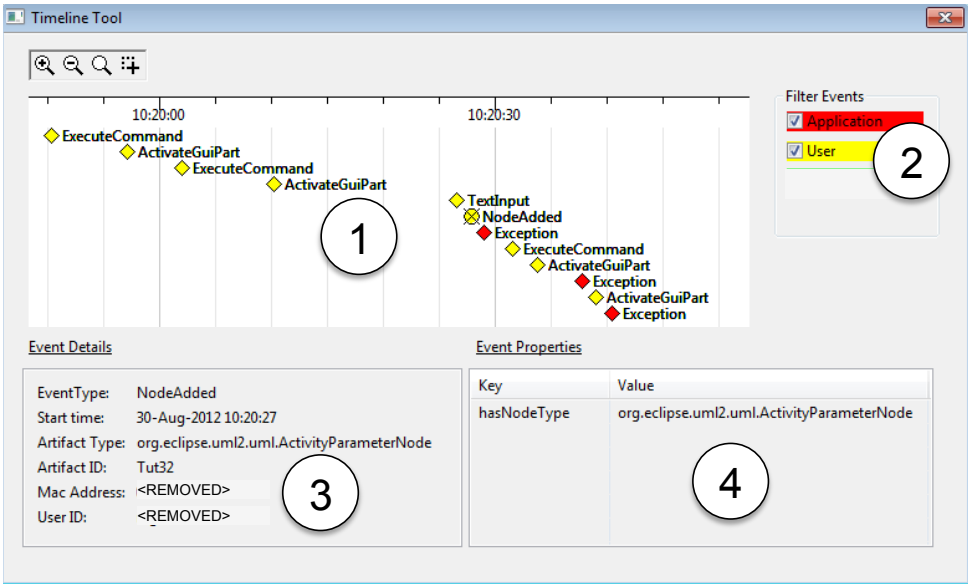


Figure 4.22: Visualization of Event Sequence in Timeline Tool (Source: [152])
1: Chronological timeline view, 2: Event type filter, 3: General event properties, 4: Event properties specific for selected event

Event Sequence Visualization in Timeline Tool

To facilitate developers to inspect and analyze monitored data, this component visualizes event sequences as shown in Figure 4.22. Developers can load a particular event sequence and see it graphically on a timeline. They can navigate the event sequence zooming in and out. Furthermore, they can filter event types, i.e. apply a filter to display or hide types of events. When developers are interested in particular event of the timeline, they can select it and inspect its properties. This functionality is implemented in the Timeline Tool, a Java application which connects to the Database Data Store, retrieves monitored interactions traces, and visualizes them.

Bug Tracker Adapter

To integrate information extracted by MALTASE in the workflow of developers, this component implements functionality to create a bug report and inject it in an existing bug tracker. This component is used when a data analysis detects a failure. Figure 4.23 sketches how the integration works: An analysis component retrieves monitored data from the server data store and analyzes it. If the analysis detects a new failure, information about the failure is pushed to the component *Bug Track Adapter*. This component creates a bug report which contains information about the failure as well as the user interactions which preceding the failure. This bug report is injected to the bug tracker running on another device via a HTTP or TCP/IP connection. Currently the MALTASE framework supports the Trac bug tracker[7].

[7]http://trac.edgewall.org/ (Accessed Jan 2015)

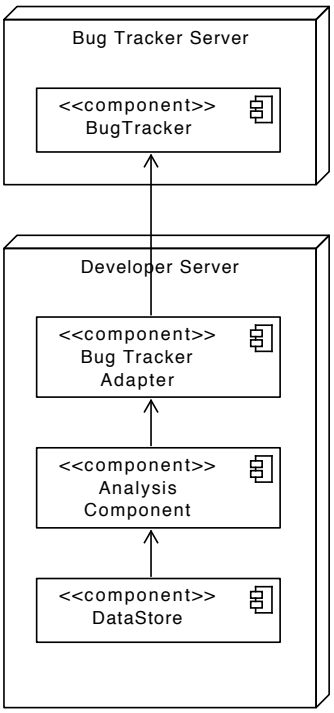


Figure 4.23: Bug Tracker Integration
Arrows denote data flow between components

4.5 MALTASE Usage Scenarios

MALTASE is agnostic to the development process used and can be employed in different phases of a software development project or during software evolution. This dissertation focuses on the use of MALTASE in software evolution, but we envision three usage scenarios for MALTASE : gathering usage information in prototype-based software development, supporting test documentation during software testing, and gathering usage information during software evolution.

In the first usage scenario, MALTASE is used to gather usage information in prototype-based software development. Hence, the scenario requires a software development process which frequently creates usable prototypes. This is the case for many agile processes which emphasize the frequent creation of testable product increments. In such a scenario, the MALTASE framework is integrated with the prototype and interactions of users testing the prototype are captured and analyzed. This allows to exploit insights gained early during development.

In the second usage scenario, MALTASE is used to document software testing activities. The MALTASE framework is integrated with a version of the target application to be tested. While testers interact with the target application during testing, their interactions are captured and analyzed by MALTASE . When testers trigger a failure, they don't have to manually describe their interactions that led to the failure because MALTASE already monitored them in the background (cf. MALTASE-based failure reproduction in Section 5.1).

In the third usage scenario, MALTASE is used to collect usage information during software evolution. The MALTASE framework is integrated with the target application and deployed together with it to users in the field. MALTASE automatically monitors and analyzes user interactions and sends this information back to developers. This scenario constitutes a continuous, automated feedback mechanism between users and developers. The usage information gained is used in software evolution and to improve the target application.

Several agile development practices and corresponding tools can be viewed as enablers for the MALTASE approach. The practices of continuos integration and continuous delivery ensure that a usable prototype of the target application exists at any time. This fits well with MALTASE because at least a usable prototype is necessary to monitor user interactions. Infrastructure tools for continuous delivery allow the deployment of application releases to users and collect data from users. Examples of such infrastructures are application distribution platforms such as TestFlight[8] or Hockey App[9]. Similarly, several companies have introduced automated crash reporting tools such as Microsoft Windows Error Reporting [63] or Apple Crash Reporter [10]. These tools collect crash data on a user device and send it to a developer server but usually do not capture user interactions. Hence, we argue that they should be extended by the collection of user interactions data to improve the tools and to implement the MALTASE approach.

4.6 Related Work

The purpose of the MALTASE framework is the acquisition of usage information by monitoring and analyzing high-level user interactions. This section discusses related work in two directions, namely the monitoring of user interactions and their analysis and exploitation. To focus the discussion, we do not review related work targeting web applications or mobile applications as well as approaches monitoring plain usage, i.e. no user interactions, or code execution.

Recent empirical studies found that usage information is important for developers. Begel and Zimmermann [15] collected a list of prioritized questions which are important for developers and should be tackled by data scientists. The two top rated questions were usage-related: “How do users typically use my application?” and “What parts of a software product are most used and/or loved by customers?”. Similarly, Buse and Zimmermann [26] present information needs during software analytics. They describe the decision scenario “understanding customers” which “leverages information about customer behavior when making decisions”, e.g. to “understand how a user is using our product” or whether users are “performing tasks we expect”. Redmiles [143] argues for the collection of usage information and describes a roadmap for human-centered software development with the goal to evolve a software application through data-driven feedback. His roadmap consists of the four main activities Observe, Use, Design, and Review. Observe denotes observing users and their organizational context in the field using activity theory to investigate

[8] https://www.testflightapp.com/ (Accessed Jan 2015)
[9] http://hockeyapp.net (Accessed Jan 2015)
[10] https://developer.apple.com/library/mac/#technotes/tn2004/tn2123.html (Accessed Feb 2015)

the usage context and derive requirements. Use denotes deploying an instrumented application and collecting usage data. Finally, Review denotes maintenance of a knowledge base with knowledge about an application and its use and exploiting this knowledge base during software evolution, e.g. by basing evolution decisions on this knowledge. MALTASE fits well into this roadmap and implements its Use activity. Orso [131] argues for the collection and exploitation of field data during software engineering. According to him, the complexity of software itself as well as the heterogeneity of software environments is growing. Hence, it is difficult to envision and test all possible usage environments before releasing an application and developers are often unaware of the behavior of their software in the field. Orso argues that field data should be collected to help developers overcoming the problems of increasing complexity and unpredictability. MALTASE fits well in his argumentation as it collects one type of field data, namely user interactions. Diep [38] discusses how usage data of deployed software can be collected and analyzed. She identifies three phases: (1) pre-deployment phase when instrumentation probes are inserted, (2) during-deployment phase when usage data is collected and sent to analysis servers, and (3) post-deployment phase when collected field data is analyzed. MALTASE supports all three phases as it instruments target applications using sensors (pre-deployment), collects and transfers monitored data (during-deployment), and enables analysis of monitored data (post-deployment). In contrast to the MALTASE framework, Diep focuses on monitoring code execution to improve in-house testing activities.

Monitoring User Interactions

This section presents related work also monitoring user interactions with desktop applications. User interactions can be monitored in several ways and on several abstraction levels. Developers can monitor user interactions by manually adding dedicated monitoring code to the code base of their application [179]. Frameworks like Google Analytics (which is discussed below) can be used to transfer monitored data to developer servers and analyze them. This instrumentation approach has the drawbacks that monitoring code has to be added manually and it is usually distributed throughout the code base. This distribution makes it difficult to maintain monitoring code and vulnerable to changes of the user interface [179]. In contrast, MALTASE monitors user interactions by software sensors which integrate with application or GUI toolkits. This mechanism avoids manual instrumentation throughout the code base of the target application and recompilation of the target application because of the instrumentation.

To address the drawbacks of adding monitoring code to the code base of a target application, aspect-oriented approaches have been proposed to instrument applications and monitor user interface events. Tao [177, 178, 179] monitors user action events in Java applications. He exploits the Model View Controller (MVC) architecture by observing event listeners implementing MVC via aspect oriented programming. Hartman and Bass [71] use an aspect oriented approach to monitor GUI events and additional processing information from different architecture layers. They also monitor changes to documents via the undo mechanism of the GUI toolkit. Bateman et al. [12] designed UMARA, a system which allows instrumentation without the

need of programming. Users can select widget of Java GUIs and specify which widget events should be captured. UMARA is based on aspect-oriented programming to implement the instrumentation. Similarly, Shekh and Tyerman [165] as well as Tarta and Moldovan [180] present frameworks which use aspect-oriented programming to collect user interface events and evaluate the usability of an application. In contrast, MALTASE monitors user interactions on the levels of user interface events as well as abstract interactions. MALTASE does not use aspect oriented programming but its sensors integrate via toolkit listener mechanisms with a target application. This mechanism avoids the addition of monitoring code as well as the recompilation of the target application because of the instrumentation.

User interactions can be monitored by exploiting listener mechanisms of application or GUI toolkits. The MALTASE sensors use such a mechanism to integrate with a target application. The FastFix platform [137] deploys sensors implemented as toolkit listeners to target applications. While MALTASE and FastFix use similar sensors for data collection, MALTASE extends FastFix and provides additional functionality to analyze monitored data. The Eclipse Usage Data Collector (UCD) [11] is a framework to collect usage data from users employing Eclipse-based applications. It monitors commands, actions invoked via menus or toolbars, and perspective changes. To monitor this data, Eclipse UCD uses listeners installed in the Eclipse framework. While MALTASE and Eclipse UCD use a similar mechanism to monitor user interactions, MALTASE provides functionality to analyze monitored data which is not considered by Eclipse UCD. Dostál and Eichler [41] monitor user commands in OpenOffice using a customized macro recorder. Furthermore, they log user interface events to determine the interaction style, i.e. whether a command was triggered by the main menu, the toolbar, or a hot key. MALTASE monitors user commands in a similar way but additionally monitors artifact manipulations and provides functionality to analyze monitored data.

User interactions can be monitored using model-driven instrumentation as described by Funk et al. [55]. Their approach models user interfaces in UML, identifies GUI elements to be monitored by assigning a particular UML stereotype, and generates application and monitoring code from these models. While this approach has the advantage that the instrumentation can be changed easily by changing the GUI model and re-generation of code, it is applicable only to applications which are developed using GUI models. MALTASE does not monitor user interactions using a model-driven approach.

User interactions can be monitored by instrumenting the toolkit used by the target application or the system environment on which the target application is deployed [12]. Toolkit instrumentation denotes the instrumentation of the toolkit which processes user interface events [12]. For example, Fenstermacher and Ginsburg [49] monitor user interactions indirectly by monitoring events in applications implemented using the Common Object Model COM. While the instrumentation is reusable for all applications using the toolkit, the monitored interactions are rather low-level, potentially causing problems to abstract and interpret collected data. MALTASE uses a middle way between application and toolkit instrumentation. The MALTASE sensors are installed in individual target applications, but

[11] http://eclipse.org/org/usagedata/ (Accessed Jan 2015)

depend only on a toolkit. Hence, they are independent of a particular target application and can be reused for other target applications using the same toolkit. But the toolkit itself is not instrumented.
System instrumentation denotes to use logging on operating system-level to record interface events [12]. This instrumentation is independent from the target application, but - similar to toolkit instrumentation - monitored interactions are rather low-level. For example, Alexander et al. [6] present AppMonitor, a tool which records user interface events using the SDK of Microsoft Windows. MALTASE does not monitor user interactions on operating system level.

Several approaches have been proposed to monitor artifact manipulations performed by users during their work with an application. Mylar [93, 122] monitors the interaction history of software developers with code artifacts in the Eclipse IDE. Maalej et al. [113] provide an overview of different approaches to monitor interactions of developers with software artifacts. MALTASE exploits monitored data for different purposes. Terry et al. [182] as well as Hartman and Bass [71] monitor document changes via the undo feature of the target application. The MALTASE framework describes two ways of monitoring user interactions, monitoring of undo operations and the MALTASE sensors.

Analyzing and Exploiting User Interactions

This section presents related work analyzing and exploiting monitored user interactions in software engineering or software evolution. Because the analysis and exploitation are often closely related, we discuss both together. Related work for the MALTASE applications, i.e. failure reproduction, skill detection, and use case testing, is also discussed in Chapter 5.

Monitored user interactions can be analyzed for different purposes using different analysis algorithms. Liu et al. [111] present a sequence mining approach to detect frequent episodes in user action sequences. Similarly, El-Ramly et al. present the algorithms IPM [46] and IPM2 [44] to discover interaction patterns in interaction traces. They adapt algorithms from the field of sequential pattern mining to the specifics of user interaction traces. Like these approaches, MALTASE uses sequential pattern mining techniques to detect patterns in interaction traces, but also employs other analysis algorithms depending on the framework application.

Monitored user interactions can be exploited in software evolution. FastFix [137] supports developers to reproduce failures, identify failure causes, and automatically patch applications based on monitored usage data. MALTASE extends the failure reproduction support of FastFix and provides additional ways of exploiting monitored interactions. The PORTNEUF framework [135] monitors user interactions as context information for user feedback. The interaction context of a user is used to recommend existing feedback with a similar context and to reduce duplicate user feedback. Hilbert and Redmiles [77, 78, 80] present EDEM monitoring user interface events selectively, i.e. developers have to specify their expectations and interactions to be be monitored upfront. They use monitored interactions to analyze user behavior, validate assumptions about user behavior, assess the impact of observed user behavior, and allocate development resources. MALTASE and EDEM share the goal

to provide usage knowledge to developers via usage monitoring and implement similar approaches for user interaction monitoring and data collection. But MALTASE differs from EDEM in two directions. While EDEM monitors user interactions at the level of user interface events, MALTASE additionally monitors user interactions on the level of abstract interactions. Furthermore, MALTASE addresses different exploitations of monitored user interactions in software evolution. El-Ramly and Stroulia [43] apply sequential pattern mining methods to interaction traces in order to discover interaction patterns and exploit them for user interface reengineering. Stroulia et al. [173, 174, 175] extract models of the user interface and of user tasks from system-user interaction traces and use them to migrate legacy applications to the web. Similarly, El-Ramly et al. [45] as well as Smit et al. [170] present approaches to reverse engineer use case models from monitored interaction traces. Like MALTASE, these approaches analyzes monitored user interactions but they exploit knowledge gained for other software evolution tasks. Van der Schuur et al. [191] as well as Krusche et al. [103] describe how software operation knowledge can be incorporated in software development and evolution processes. As this area is not investigated in this disseration because of scope reasons, their approach and results should be considered when introducing MALTASE in a software evolution context.

Monitored user interactions can be exploited to get insights into user behavior. Pachidi et al. [133] analyze software operation data to gain knowledge about software usage. They target four categories of knowledge: summaries of sessions and user behavior, factors influencing customer decisions, user profiles, and frequent navigation paths. To extract this knowledge, they use classification analysis, user profiling, and clickstream analysis. They share with MALTASE the goal of providing developers with insights about software usage but target different types of knowledge and use other types of analysis. Hence, their approach is complementary to MALTASE. Several researchers, e.g. Kay and Thomas [92], Terry et al. [182] and Murphy et al. [122], have studied how users use a particular application. Microsoft's Customer Experience Improvement Programme[12] collects data about command usage (e.g. paste) and feature usage (e.g. tables). This data is used to redesign the user interface, e.g. optimize the arrangement of buttons. Runtime intelligence tools, e.g. Google Analytics[13] or Trackerbird Software Analytics [14], collect data how users use a particular application. They provide insights in user behavior such as feature usage or navigation patterns, but usually do not provide direct support for software evolution. Several researchers, e.g. Kim et al. [95], Hullet et al. [83], and Gagné et al. [56] present approaches to monitor user actions in games and extract insights to improve game design. These approaches share with MALTASE the goal to provide developers with information about software usage, but usually do not provide direct support for software evolution.

Exploiting monitored user interactions to evaluate the usability of an application has a long tradition in the field of human-computer interaction. Hilbert and Red-

[12] http://blogs.technet.com/b/office2010/archive/2010/02/09/how-does-usage-data-improve-the-office-user-experience.aspx (Accessed Feb 2015), http://blogs.technet.com/b/office2010/archive/2009/11/03/data-driven-engineering-tracking-usage-to-make-decisions.aspx (Accessed Feb 2015)

[13] http://www.google.com/analytics/ (Accessed Feb 2015)

[14] http://www.trackerbird.com/ (Accessed Feb 2015)

miles [79] provide an overview of techniques to extract usability-related information from monitored user interface events. Similarly, Ivory and Hearst [85] provide an overview of automated methods for usability evaluation. Hoiem and Sullivan [81] provide an overview of tools for computer-aided usability engineering (CAUSE), i.e. tools for collection and analysis of usability data. Lecerof and Paternò [106] evaluate user interfaces using task models and logs generated from user tests. Akers et al. [5] present backtracking analysis, a usability method which treats backtracking actions of users like undo as indicators for usability problems. Tao [179] use a grammar-based approach to detect the user's task based on monitored user interface events. In contrast to these approaches, MALTASE does not aim to evaluate the usability of applications but to exploit monitored user interactions in software evolution.

Monitored interactions of developers with CASE tools have been used to study developer behavior. Murphy et al. [122] study how developers use the Eclipse IDE. Similarly, the Eclipse Usage Data Collector[15] collects usage data of developers working with Eclipse. Murphy-Hill et al. [124] analyze refactoring practices of developers based on monitored developer actions. Similarly, ElectroCodeoGram [160] monitor programming actions of developers and analyze them to study programmer behavior. Hackystat Hackystat [90] analyzes developer actions and development processes in combination. Furthermore, monitored interactions of developers with CASE tools have been used to improve CASE tools and support developers in their work. Maalej et al. [113] review approaches collecting interaction data and exploiting them as database for developer recommender systems. Maalej and Happel [114] present TeamWeaver, a framework monitoring developer interactions with development artifacts to automatically capture developer knowledge and enable knowledge exchange in development teams. Similarly, Kersten and Murphy [93] monitor developer interactions with code artifacts to create a task context, i.e. a set of code artifacts relevant to the current task. This task context is used to filter information displayed in the IDE and reduce information overload. INTI [112] monitors interactions of developers with CASE tools to detect the current intention of a developer and integrate different CASE tools. In contrast to these approaches, MALTASE does not aim to study developer behavior or generate recommendations for developers but to establish a feedback channel from users to developers via monitored data. As developers are users, too, MALTASE can be applied to CASE tools.

Monitored user interactions can be exploited to assist users and adapt to the current user. Several frameworks, e.g. [37, 101], capture user interactions as part of the usage context and exploit them to enable context-aware applications, i.e. applications which are aware of their context and adapt to it. Furthermore, monitored user interactions can be exploited to support users in improving their application skills. For example, Linton et al [109], Murphy-Hill et al. [123], and Matejka et al. [120] monitor command usage and recommend users rarely used commands to improve their application skill. Similarly, monitored user interactions can be exploited to assist users by (partially) automating their tasks. For example, Liu et al. [111] present an approach to assist users of Microsoft Word to format text or apply consistent formatting styles based on frequent episodes in interaction traces. MALTASE does

[15] http://eclipse.org/org/usagedata/ (Accessed Feb 2015)

not address such exploitation of monitored interactions. But we hypothesize that interactions monitored by MALTASE can be used for such purposes.

User interactions are one type of software operation knowledge. Van der Schuur et al. [190] present a reference framework for utilization of software operation knowledge. They define four types of software operation knowledge, namely performance, quality, usage, and end-user feedback. Additionally, they describe how such knowledge can be integrated into software evolution processes. Furthermore, Kristjánsson and van der Schuur [102] survey industrial tools for the acquisition of software operation knowledge. MALTASE fits into this framework as it investigates user interactions as one type of software operation knowledge in detail and demonstrates exploitation of user interaction knowledge.

4.7 Chapter Summary

This chapter described the MALTASE framework in detail. It listed requirements for a framework monitoring, analyzing, and exploiting user interactions in software evolution: monitoring high-level user interactions because of their semantic, easy integration of sensors in target applications to minimize instrumentation effort, introduction of an acceptable performance overhead to avoid hindering users in their work, maintaining user privacy to ensure user acceptance, visualization of monitored data and analysis results to enable developers to inspect and exploit usage knowledge, and the integration of the framework into existing tool chains to avoid tool islands. Also, this chapter presented a conceptual model of user interactions, users, software applications and usage contexts.

Furthermore, this chapter presented the architecture of the MALTASE framework consisting of four layers. The Monitoring & Information Extraction layer is responsible for monitoring user interactions. MALTASE monitors high-level user interactions such as commands and artifact manipulations to capture semantics. The Monitoring & Information Extraction layer uses two mechanisms to monitor user interactions: the MALTASE sensors and the Undo History Extractor. The MALTASE sensors hook as listeners into application or GUI toolkits. Hence, they are reusable for other applications based on the same toolkits and do not require dedicated monitoring code or recompilation of the target application. The Undo History Extractor extracts the interaction history of the undo feature and enables monitoring with minimal runtime overhead because this information is captured anyway. The components of the Monitoring & Information Extraction Layer are the only components of the MALTASE framework which depend on the target application or its toolkits. The Data Storage & Transfer layer is responsible for persisting monitored data and transmitting them to a developer server. For this purpose, this chapter presented a taxonomy of application events and user interactions, a database layout, a XML file format, as well as a client-server communication mechanism based on TCP sockets. The Processing & Analysis layer is responsible for processing and mining monitored data. For this purpose, this chapter described preprocessing operations such as filtering, sessionizing, sorting, or aggregating events. Further, it defined analysis functionality such as feature extraction, sequential pattern mining, or classification. The Presentation & Integration layer is responsible to present monitored data and analysis results to

developers and integrate this information into developer tools. For this purpose, it contains a component to visualize monitored user interactions as well as component to generate bug reports and inject them into a bug repository.

We presented three usage scenarios of the MALTASE framework: software evolution, software testing, and prototype-based software development. In the software evolution scenario, an instrumented version of the target application is deployed to end users and used to collect field usage data. In the software testing scenario, an instrumented version of the target application is deployed to testers and used to document the testing activities. And in the prototype-based software development scenario, an instrumented prototype is deployed to user representatives and used to collect early feedback.

Finally, we discussed related work for the MALTASE framework. We found that many researchers investigated automated usability evaluation based on monitored user interactions as well as usage of CASE tools by software developers. MALTASE differs from these areas as it exploits monitored user interactions during software evolution. Furthermore, several runtime intelligence tools are used in practice which monitor software usage and provide insights into user behavior. MALTASE complements those tools by focusing on extraction of usage information relevant during software evolution and its exploitation.

Overall, the MALTASE framework implements an indirect communication channel between users and developers via usage monitoring. This communication channel requires no effort from user beyond their agreement of being monitored. Developers can inspect monitored data and analysis results to get insights into user behavior, exploit this knowledge in evolution tasks, and base evolution decisions on it. Hence, MALTASE addresses the communication gap between developers and users and provides developers with usage knowledge which is helpful during software evolution.

Chapter 5

Framework Applications

This chapter describes three ways to employ the MALTASE framework to gain usage knowledge for developers during software evolution. We call these ways "framework applications". The first framework application "MALTASE-based failure reproduction" (Section 5.1) exploits monitored user interactions preceding failures as reproduction steps during bug fixing. The second application "MALTASE-based skill detection" (Section 5.2) classifies users according to their skills based on monitored user interactions. And the third application "MALTASE-based Use Case Testing" (Section 5.3) compares monitored user interactions with use case steps to test whether users employ an application as expected by developers.

5.1 MALTASE-based Failure Reproduction

This section describes MALTASE-based failure reproduction, an approach monitoring user interactions preceding failures and presenting them to developers during bug fixing as reproduction steps. We hypothesize that this information helps developers to reproduce and fix bugs. This approach has been published in Roehm et al. [152]. An empirical evaluation of the approach is presented in Section 6.3.

5.1.1 Motivation

> "93% of 1,477 participating software developers encounter problems due to missing knowledge when reproducing failures weekly, 70% daily." Maalej et al. [117]

Reproducing failures allows software developers to verify that a problem exists and constitutes a first step towards identification of the failure cause and fixing the bug causing the failure. To reproduce a particular failure, developers need information about reproduction steps, i.e. the steps necessary to trigger the failure, and information about the failure environment, i.e. the setting in which the failure occurs [201]. Because developers are usually not present when users employ an application in the field, they do not have first hand information about reproduction steps and failure environments. Correspondingly, participants of the problem case study presented in Chapter 3 were interested in failure reproduction steps as one type of usage information. But because such information is rarely available to developers, they frequently face problems when reproducing failures. In a survey [117], 93% of 1,477 participating developers reported that they face problems due to missing knowledge

when reproducing failures at least weekly, 70% faced such problems on a daily basis. Missing information of developers about failure reproduction steps and failure environments constitutes a sub-problem of missing usage information.

Developers usually use two ways to acquire information about reproduction steps and failure environments: bug reporting and collection of field data. During bug reporting, users who experienced a failure report it together with reproduction steps and information about the failure environment. Research has shown that this approach has some challenges because bug reports submitted by users often do not contain reproduction steps or reported reproduction steps are wrong or incomplete [105, 203]. Alternatively, developers can instrument applications and collect field data, i.e. data about the runtime behavior and runtime environment of deployed programs [131]. Such an approach usually generates a big amount of trace data and developers have to be supported to extract information about failure reproduction steps and failure environment.

To address the problem of missing information about failure reproduction steps, we apply the MALTASE framework to monitor user interactions preceding failures and provide them to developers during bug fixing. We call this approach MALTASE-based failure reproduction. It is based on the observation that reproduction steps for interactive applications are usually the user interactions performed before a failure occurs. Furthermore, monitoring high-level user interactions reduces the size of monitored traces and potentially enables developers to manually inspect monitored interaction traces and reproduce failures based on this information. MALTASE-based failure reproduction continuously monitors user interactions as well as the occurrence of failures. Upon failure occurrence, it automatically creates a bug report which contains information about the failure as well as the history of user interactions preceding the failure.

5.1.2 Approach

Figure 5.1 shows an overview of MALTASE-based failure reproduction which consists of five main steps): monitoring user interactions and failure events, mapping monitored events to a taxonomy, transferring failure information and monitored interactions to a developer server, automated generation of bug reports, and visualization of monitored interaction traces.

Monitoring User Interactions and Failure Events MALTASE-based failure reproduction uses the MALTASE framework to monitor user interactions and failure events. While a user is interacting with a target application, the MALTASE Sensors capture these events and store them temporarily in the File Data Store on the user device. The MALTASE Sensors monitor user interactions on a high-level of abstraction such as user commands or artifact manipulations. Furthermore, the MALTASE Exception Sensor monitors the occurrence of exceptions. A detected exception triggers the extraction of monitored user interactions from the File Data Store and their transfer to a developer server.

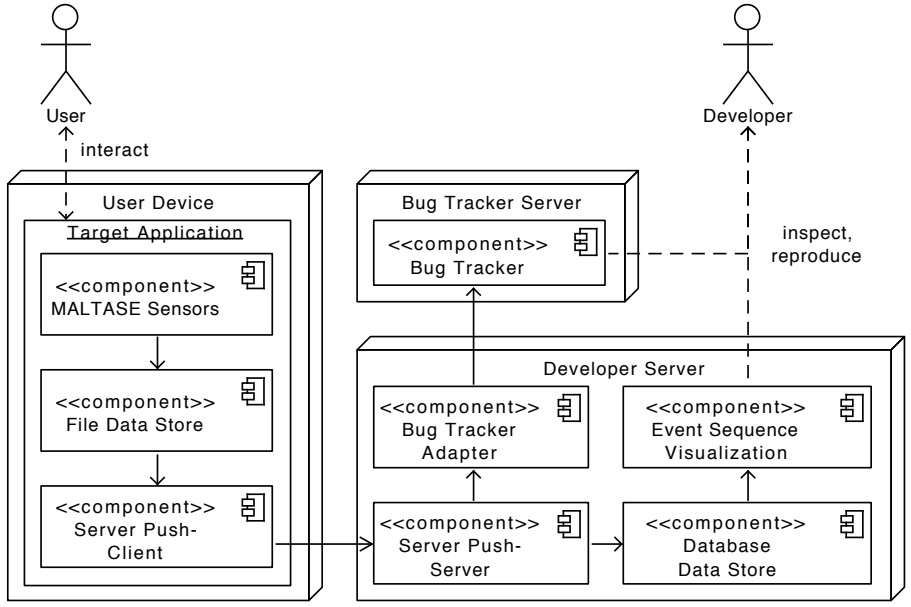


Figure 5.1: Overview of MALTASE-based Failure Reproduction
Arrows denote data flow
Based on Figures 4.12 and 4.13

Mapping to Taxonomy of User Interactions and Application Events To assist developers to comprehend monitored user interactions, monitored user interactions are mapped to a taxonomy of user interactions which is part of the MALTASE ontology. It is shown in Figure 5.2 on the right side. This mapping provides semantics to monitored interactions via the ontology, i.e. the ontology defines the meaning for each particular interaction. The mapping is performed by the MALTASE Sensors by assigning ontology descriptors as event types for detected events.

Transfer of Monitored Data to Developer Server The detection of an exception by the Exception Sensor triggers the extraction of monitored interactions from the File Data Store, the bundling with additional failure information, and the transfer to a developer server. This functionality is implemented by the MALTASE components Server-Push Client and Server-Push Server. When information about a specific failure arrives at the developer server, it is processed in two ways: A bug report is created automatically and injected in the bug tracking system. Furthermore, monitored data is stored in the Database Data Store. Developers can inspect monitored data in the Database Data Store using the Timeline Tool which is part of the component Event Sequence Visualization.

Generation of Bug Reports With Interaction History To inform developers about failures and support developers to reproduce them, bug reports with monitored user interactions preceding a failure are generated automatically. Upon arrival at the developer server, failure information is pushed to the Bug Tracker Adapter. This component creates a new bug report, adds the interaction history, and injects it in the bug tracking system.

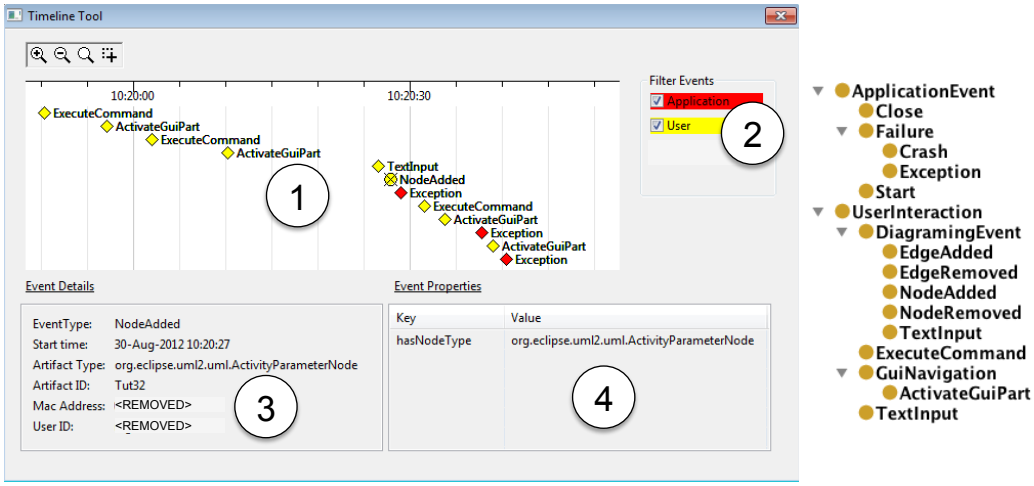


Figure 5.2: Visualization of Event Sequence in Timeline Tool (Left) and Taxonomy of User Interactions and Application Events (Right)
Events in trace correspond to event types in taxonomy
Source: Figures 4.19 and 4.22

Visualization of Interaction Traces To allow developers to inspect monitored data, the Timeline Tool (which is part of the component Event Sequence Visualization) visualizes sequences of monitored user interactions and application events. It is shown in Figure 5.2. Developers during bug fixing can inspect the user interactions preceding failures and exploit this information to reproduce failures.

As with every approach, MALTASE-based failure reproduction has prerequisites, advantages, and limitations. MALTASE-based failure reproduction requires an interactive target application whose main source of non-determinism are user interactions. To use the MALTASE sensors, the target application has to be implemented using the Eclipse RCP framework as well as the Eclipse Modeling Framework. Otherwise, new sensors for the target application under study have to be implemented while the other parts of the MALTASE framework can be reused.
The fact that the MALTASE framework monitors user interactions at a high level of abstraction contributes several advantages to MALTASE-based failure reproduction: Because monitored user interactions are assigned meaning by mapping them to a taxonomy, developers can understand user behavior. Because user interactions are monitored, developers can discuss them with users (which they could not in case of code execution monitoring). Because high-level monitoring usually captures a smaller number of interaction events, developers are enabled to manually inspect and analyze monitored traces. Because user interactions are captured automatically, users do not have to remember or describe their interactions. Furthermore, the automated generation of bug reports contributes advantages to MALTASE-based failure reproduction: Developers are informed proactively about field failures, bug reports contain monitored user interactions preceding failures as reproduction steps, and the approach is integrated into the developer workflow.
MALTASE-based failure reproduction has some limitations which should be considered when applying it. It does not deal with noisy interactions, i.e. interactions

which are not necessary for failure reproduction. Furthermore, it focusses on the provision of reproduction steps and does not address reproduction of the failure environment. Hence, it should be complemented with approaches tackling this limitation or capture information about the failure environment in addition to user interactions. Additionally, MALTASE-based failure reproduction addresses user input as source of non-determinism while there are other types of non-determinism like thread scheduling or network traffic [201]. Furthermore, privacy issues arise because monitoring user interactions might capture sensitive data.

5.1.3 Related Work

Many approaches collect runtime data from deployed software. Tucek et al. [186] re-execute applications to collect additional failure information. Artzi et al. [9] generate and execute multiple tests to reproduce a given failure. Liblit et al. [108] monitor code execution of deployed software and analyze this data to isolate bugs automatically. In contrast to MALTASE-based failure reproduction, these approaches monitor code execution. Several tools for automated crash reporting exist which report crashes and information about the crash environment via the Internet, e.g. MS Windows Error Reporting [63], Apple Crash Reporter[1], Crashlytics[2], Hockeyapp[3], or Testflight[4]. These reporting tools collect information about system state when a failure occurred such as thread states or memory dumps but do not capture user interactions. Microsoft Problem Steps Recorder[5] allows users to record their interactions and captures a screenshot for every user interaction. In contrast to MALTASE-based failure reproduction, users have to start the recording proactively and user interactions are recorded on GUI event level.

Capture/replay approaches have been proposed to capture and replay user events, application events, and system events at different levels of abstraction (e.g. [16, 24, 64, 91, 132]). Clause and Orso [30] present an approach to debug field failures by recording file system and stream actions, minimizing captured traces, and replaying them. Herbold et al. [75] and Steven et al. [172] capture application events triggered by user interactions and enable developers to replay them. Because these tools have to ensure replay of captured traces, they monitor events at a lower level of abstraction than the MALTASE sensors.

Several approaches to reproduce field failures use a combination of field data collection and in-house execution synthetization: BugRedux [88] is a general framework to collect failing field executions and to synthesize executions which trigger the same failure. F^3 [89] extends BugRedux to synthesize passing and failing executions and exploits them to identify potentially faulty program entities. MIMIC [204] extends F^3 and compares a model of correct behavior to failing executions, identifying violations of the model as potential explanations for failures. In contrast to MALTASE-based failure reproduction, those approaches also operate on code level.

[1] https://developer.apple.com/library/mac/technotes/tn2004/tn2123.html (Accessed Feb 2015)
[2] https://crashlytics.com/ (Accessed Feb 2015)
[3] http://hockeyapp.net (Accessed Feb 2015)
[4] https://www.testflightapp.com/ (Accessed Feb 2015)
[5] http://technet.microsoft.com/en-us/windows/dd320286.aspx (Accessed Feb 2015)

Several approaches have been proposed to automatically identify user input triggering failures. For example, Clause and Orso [31] present Penumbra which identifies failure-relevant inputs using dynamic tainting. Similarly, Zeller and Hildebrandt [202] present the Delta Debugging algorithm which minimizes captured user input or source code to the subset relevant for failure reproduction. MALTASE-based failure reproduction does not apply such minimization techniques.

Another way to identify failure-inducing program input is to generate input values and test whether they trigger a failure. For example, Kifetew et al. [94] describe a search-based failure reproduction approach which uses genetic programming to generate program inputs triggering failures. Cao et al. [27] combine a capture/replay approach with input generation. They record only the hard-to-resolve functions at runtime and generate failure-inducing input using captured runtime data. In contrast to these approaches, MALTASE-based failure reproduction captures interactions of real users and allows developers to inspect user behavior preceding failures.

Zimmermann et al. [203] propose to educate bug reporters to provide information about reproduction steps when composing a bug report.

5.2 MALTASE-based Skill Detection

This section describes MALTASE-based skill detection, an approach classifying users according to their skill based on monitored user interactions. We hypothesize that skill information supports developers to evolve an application and its help system according to user needs. Furthermore, skill information can be used by intelligent applications to adapt to the current user. An empirical evaluation of MALTASE-based skill detection is presented in Section 6.4.

5.2.1 Motivation

> "Developers face the task of writing software for millions of users (at design time) while making it work as if it were designed for each individual user (only known at use time)." Fischer [53]

To address the challenge described by Fischer [53] and optimize a software application for the needs of individual users, developers need information about users and usage contexts. Therefore, methods have been developed in the field of requirements engineering (e.g. observations and interviews) and software development (e.g. participatory design) to involve users during software design and elicit such information from them. But it is difficult to anticipate all types of users and usage contexts before the deployment of an application. To address the problem of missing information about software users, we apply the MALTASE framework to monitor user interactions and extract information about individual users from their interactions.

We focus on detecting user skill information because skills have a high impact on performance when solving tasks with an application (Ghazarian and Noorhosseini [61]) and call this approach MALTASE-based skill detection. We hypothesize that skill information for specific users and the user population in total can be exploited by developers for three purposes. First, skill information supports developers

in their evolution decisions. For example, the implementation of advanced features is only reasonable if a sufficiently large group of expert users exists. Second, knowledge about user skills constitutes important context information when interpreting other types of usage information. For example, a problem described in a bug report submitted by a novice user might be no bug but originate from the inexperience of the user. And third, information about the current users' skill allows an application to adapt to the current user to improve performance (Trumbly et al. [185]). Developers usually do not have user skill information, potentially because of communication gaps and the nature of skill which is not directly observable and difficult to assess. Because the skill level of the current user is usually unknown, developers usually base their decisions on assumptions about "the typical user which does not exist" (Fischer [53]). Furthermore, most applications use a uniform, static user interface and help system for all users.

Nielsen [127] identifies three types of user experience: computing experience (ability with regard to computer use in general), system experience (ability with regard to a particular system or software application), and domain experience (ability with regard to a particular task domain). In contrast to Nielsen, we use the term "skill" instead of "experience" because we focus on the ability of users to accomplish tasks ("skill") and not how this ability is acquired ("experience") or more abstract concepts ("knowledge", "expertise"). Researchers have already investigated the detection of computing experience from monitored user interactions (Ghazarian and Noorhosseini. [61]). Hence, we focus on the detection of system experience and domain experience and call them "application skill" and "domain skill", respectively. Following Nielsen, we hypothesize that users need mainly two types of skill to accomplish a given task using a software application: domain skill to know *what* they have to do and application skill to know *how* to employ a particular software application to accomplish "this what". For example, business people need to know specific formats of formal business letters and *how* to use Microsoft Word or LaTeX to produce the corresponding format. Furthermore, we hypothesize that discriminating between application skill and domain skill is helpful to provide context-specific help and to adapt to the current user. For example, it allows to distinguish if the source of a user problem is the users unfamiliarity with the task domain or with the current application. Consequently, an intelligent application can "explain" its features (in case of low application skill) or present training material about the task domain (in case of low domain skill).

5.2.2 Approach

Figure 5.3 showls an overview of MALTASE-based skill detection which consists of four main steps: monitoring user interactions, extraction of classification features from monitored interaction traces, learning skill classifiers, and skill detection.

Monitoring User Interactions MALTASE-based skill detection uses the MALTASE Sensors to monitor user interactions while a user is interacting with a target application. Monitored interactions are stored temporarily on the user device in the File Data Store and are transferred to a developer server for analysis purposes.

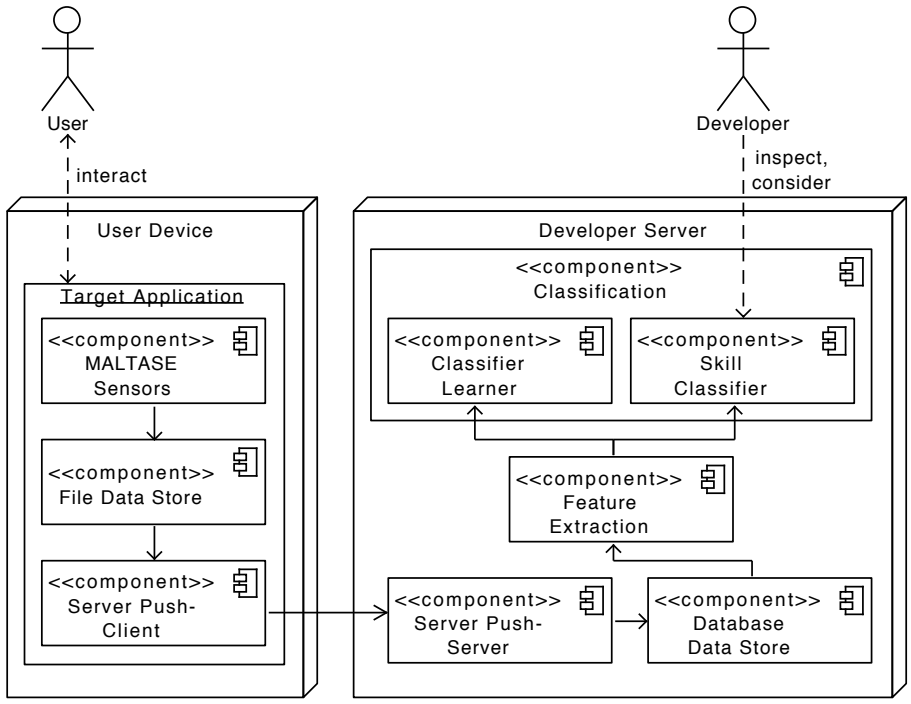


Figure 5.3: Overview of MALTASE-based Skill Detection
Arrows denote data flow
Based on Figures 4.12 and 4.13

Extraction of Classification Features To aggregate monitored data and extract features necessary for classification algorithms, the component Feature Extraction derives classification features from interaction traces. Examples of classification features are the number of interactions per minute, the usage of hot keys, or the average length of a break between two interactions. Extracted classification features are used in two ways. The component Skill Classifier uses them to classify the skill level of a user. Furthermore, the component Classifier Learner uses them to learn classifier models.

Learning of Classifiers for User Skill Before classifiers can be used to detect user skill, classifier models have to be learned. This is the task of the component Classifier Learner. It learns skill classifiers using a supervised learning approach. More specifically, it learns a decision tree classifier model [141] from sets of classification feature values together with skill labels. Learning is performed offline, i.e. on the developer server, and learned classifier models are used by component Skill Classifier to detect user skill levels. For each type of skill a separate classifier model is learned.

Detection of User Skill Finally, component Skill Classifier uses decision tree classifier models [141] learned by component Classifier learner to detect user skill. Then, developers can inspect skill information and consider it in their evolution decisions. While the user interaction monitoring has to be done at runtime, feature extraction

and skill classification can be performed either online on the user device or offline on a developer server. Online classification has the advantage that detected user skill information can be used for user interface adaptation but it introduces a performance overhead for the user. Offline classification minimizes performance overhead but it requires to transfer monitored data to a developer server. Furthermore, its classification results cannot be used for user interface adaptation. Developers have to choose between online and offline classification depending on their situation.

As with every approach, MALTASE-based skill detection has prerequisites, advantages, and limitations. To successfully employ MALTASE-based skill detection, it must be possible to detect user skill based on classification features extracted from monitored user interactions. To reuse the MALTASE sensors, the target application has to be implemented using the Eclipse RCP framework as well as the Eclipse Modeling Framework. Otherwise, new sensors for the target application have to be implemented while the other parts of the MALTASE framework can be reused.
MALTASE -based skill detection is performed in the background and does not require users to spend effort or time. Furthermore, the monitoring of user interactions at a high abstraction level potentially enables the detection of domain skill which can't be detected based on low-level interactions.
MALTASE-based skill detection has some limitations which should be considered when using it. As the concept of skill is difficult to measure and generalize, the accurracy of MALTASE-based skill detection has to be carefully investigated. Furthermore, user skill usually changes over time when a user becomes familiar with a software application or a task domain. This fact is not considered by MALTASE-based skill detection. Future work could extend the current approach and address this dynamic. Additionally, ethical questions as well as privacy issues arise during detection, processing, and storage of skill information. These issues have to be addressed by ensuring that skill information can only be accessed by authorized people and that it is used for legal purposes only.

5.2.3 Related Work

Several researchers studied the automatic detection of user skills from monitored user interactions. Ghazarian and Noorhosseini [61] detect system and task skill from from low-level, high frequency user interactions such as mouse and keyboard interactions for a paint application. Ghazarian and Ghazarian [60] detect task skill based on pauses between mouse and keyboard interactions for a paint application. Hurst et al. [84] detect task skill from mouse and menu interactions for an image editing application. Vaubel and Gettys [192] predict general user skill from command frequencies and help requests for a word processing application. Beale et al. [13] detect general user skill from text commands for functional programming tool. Nagasaki and Azuma [125] detect task skill for a word processor and a modeling application based on general user interactions. Sao Pedro et al. [158] present an approach to detect scientific inquiry skill of students from activity logs using machine learning detectors. While these approaches detect user skill from monitored user interactions similar to MALTASE-based skill detection, they do not discriminate

between application skill and domain skill. Furthermore, they do not discuss the exploitation of skill information during software evolution.

MALTASE-based skill detection creates a model of individual users based on their detected skills. Building a model of users is usually called "user modeling" and this activity is investigated by several researchers, mainly in the area of human-computer interaction. Fischer [53] reviews user modeling techniques in human-computer interaction and defines "user models" as "models that systems have of users". Similarly, Bezold and Minker [17] provide an overview of user modeling for interactive systems with the goal of user adaptation. The Springer journal "User Modeling and User-Adapted Interaction"[6] and the conference series "Conference on User Modeling, Adaptation and Personalization"[7] are dedicated venues for research about user modeling and user adaptation.

5.3 MALTASE-based Use Case Testing

This section describes MALTASE-based use case testing, an approach comparing monitored user interactions with use case steps. We hypothesize that information about differences between both supports developers to detect software improvements and update the use case documentation. The idea of MALTASE-based use case testing has been published in Roehm et al. [151]. This section describes a solution which maps the problem to the process mining domain and reuses process mining techniques to implement the detection of differences between monitored user interactions and use case steps.

5.3.1 Motivation

As described by Norman [129], developers construct a conceptual model of software usage and implicitly assume that the user's model is identical to their model. Because of communication gaps between developers and users, the conceptual model of developers is rarely tested and developer assumptions about user behavior rarely corrected if they are wrong. Wrong developer assumptions about user behavior and corresponding design decisions can lead to low software quality and usability.

MALTASE-based use case testing addresses this problem and automatically compares developer assumptions about user behavior to monitored user behavior. To automatically compare developer assumptions, a representation of them is necessary. MALTASE-based use case testing uses the steps of use cases, also known as "flow of events", as representation of developer assumptions about user behavior because use cases are usually written by developers and "describe the behavior of a system as seen from a user's point of view" [23]. Use case steps are compared to monitored, high-level user interactions. This comparison allows to "test" the use case documentation of an application, i.e. to determine whether users use an application as described in the use case documentation.

[6] http://rd.springer.com/journal/11257 (Accessed Feb 2015)
[7] http://www.um.org/conferences (Accessed Feb 2015)

The following example illustrates the idea of MALTASE-based use case testing. We consider the development of an online banking application. Before implementing the application, developers talked to banking customers and identified the use case "Money Transfer" consisting of the user steps Login, Start bank transfer, Enter recipient, Enter amount, Submit transaction, and Logout. After the requirements engineering phase, developers designed, implemented, and deployed then the application. Nor, users use the application to transfer money from one bank account to another one. All user interactions are monitored anonymously. The comparison of use case steps and monitored user interactions detects that users frequently forget to logout. This difference is presented to the developers of the banking application with the request to decide how it should be handled. As unterminated user sessions are a security threat, they decide to add an automatic logout feature which terminates a user session after a certain time of inactivity.

We argue that differences between use case steps and user interactions can be interpreted in different ways. Developers have to analyze them, decide about their severity, and derive corresponding actions. We see the following four ways for developers to treat a detected difference. First, the difference is irrelevant and can be ignored. Second, the difference indicates a way to improve the application (as illustrated by the example). Similarly, the difference can indicate a usability issue which developers want to resolve. Third, the difference triggers an update to the use case documentation. This is the case when monitored user behavior reflects a valid use of the application which is not yet documented as a use case. And finally, the difference indicates improvements of the help system or training programs. For example, if users deviate frequently from steps of a particular use case but it is necessary to follow these steps, the help system or training program can be adjusted to provide users with help and training regarding the use case. Overall, detected differences improve the understanding of developers about user behavior because detected differences identify wrong developer assumptions. A detected difference might trigger further investigation by e.g. user interviews or a usability lab study.

5.3.2 Approach

Figure 5.4 shows an overview of MALTASE-based use case testing which consists of four main steps: monitoring user interactions, mapping to the process mining domain, detection of differences using process mining, and visualization of traces and detected differences.

Monitoring User Interactions MALTASE-based use case testing uses the MALTASE Sensors to monitor high-level interactions of a user with a target application. Monitoring user interactions at a high abstraction level is important to ensure a similar abstraction of monitored user interactions and use case steps. Monitored interaction traces are temporarily stored in the File Data Store on the user device, transferred to a developer server, and permanently stored in the Database Data Store of the developer server.

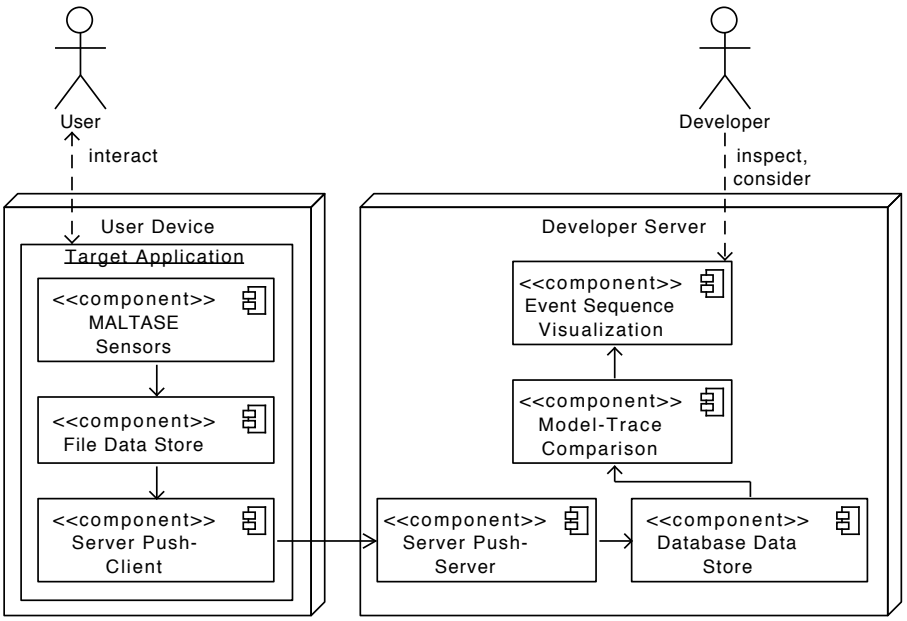


Figure 5.4: Overview of MALTASE-based Use Case Testing
Arrows denote data flow
Based on Figures 4.12 and 4.13

Mapping to Process Mining As use case steps describe expected sequences of user interactions, they constitute a model of expected user behavior. In contrast, monitored user interactions represent real user behavior in form of interaction traces. The goal of MALTASE-based use case testing is to compare both and detect differences, i.e. deviations of real user behavior from expected user behavior. This situation corresponds to the problem statement of conformance checking in process mining: "conformance checking compares an existing process model with an event log of the same process. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa" [189]. Therefore, MALTASE-based use case testing maps the problem of detecting differences between use case steps and monitored user interactions to the process mining domain and uses process mining techniques to perform the comparison. Traces of monitored user interactions are used directly. The user steps of the flow of events of a particular use case are automatically transformed to a Petri net. It is assumed that the flow of events consists of a certain number of use case steps with a given order. Each use case step is represented as a transition in the Petri net. Furthermore, two transitions representing consecutive use case steps are connected through a place.

Difference Detection Using Process Mining MALTASE-based use case testing uses process mining algorithms to detect differences between monitored user interaction and use case steps. More specifically, it uses conformance checking by token replay [189]. This algorithm requires an event log and a process model as input and "replays" the event log on top of the process model, i.e. it checks whether it is possible to fire transitions in the Petri net in the order indicated by the event log. The result of the replay are discrepancies between the log and the model. There

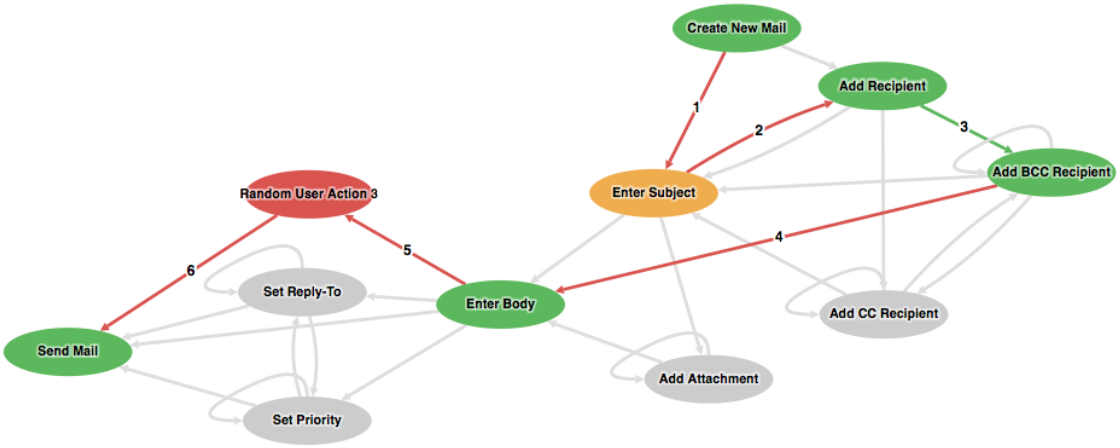


Figure 5.5: Visualization of Interaction Trace and Detected Differences
Edge numbers indicate order in trace, Red edges indicate unexpected order of interactions, Red nodes indicate additional interactions, Yellow nodes indicate "out of order" interactions

are two types of discrepancies: User interactions in the log which are missing in the model and represent user interactions not foreseen by the model at the current position ("move on log"). And user interactions in the model which are missing in the log and represent expected user interactions which were not performed by the user at the current position ("move on model"). MALTASE-based use case testing performs difference detection using the ProM process mining tool[8].

Visualization of Interaction Traces and Detected Differences To help developers inspect detected differences, the component Event Sequence Visualization visualizes a single interaction trace and its deviations from the use case steps. Figure 5.5 shows this visualization for the exemplary use case of composing an e-mail. The expected order of use case steps is Create new mail, Add recipient, Enter subject, Enter body, and finally Send mail. Developers can inspect the order of user interactions in the trace via numbered edges. A red edge represents two consecutive interactions which do not correspond to the flow of events. Similarly, red nodes represent monitored user interactions which are not contained in the flow of events and yellow nodes represent monitored interactions performed in an unexpected order.

As with every approach, MALTASE-based use case testing has prerequisites, advantages, and limitations. As it targets use cases, a use case documentation of the target application is required. To reuse the MALTASE sensors, the target application has to be implemented using the Eclipse RCP framework as well as the Eclipse Modeling Framework. Otherwise, new sensors for the target application have to be implemented while the other parts of the MALTASE framework can be reused.
MALTASE-based use case testing has the advantage of reusing established process

[8]http://processmining.org/ (Accessed Feb 2015)

mining technology which enables the detection of additional interactions and missing interactions in monitored interaction traces.
MALTASE-based use case testing has three limitations. It assumes that use case steps and monitored interactions are described at the same level of abstraction. We argue that the monitoring of high-level user interactions by the MALTASE sensors justifies this assumption. For example, all user interactions in the e-mail example in Figure 5.5 can be monitored by capturing user interface events for the corresponding text fields or buttons. If user interactions are monitored at a lower abstraction level than use case steps, an abstraction has to be performed to increases the abstraction level of them. Furthermore, MALTASE-based use case testing assumes that an interaction trace corresponds to one use case and that this use case is known. This information can be gathered by manually sessionizing interaction traces and identifying the use case of the trace. Alternatively, users can be asked to perform one particular use case and monitor their interactions. Future work could investigate the automation of the sessionization and use case detection. In addition, MALTASE-based use case testing assumes a strict order of use case steps. Future work could investigate how this strict constraint can be softened.

5.3.3 Related Work

Several researchers proposed approaches to automatically compare monitored and expected user behavior. Paternò et al. [106, 139] compare monitored user interactions with task models which represent expected user behavior. Feuerstack et al. [51] evaluate the usability of multimodal user interfaces by comparing monitored user behavior to user interface models. Girgensohn et al. [62] as well as Hilbert and Redmiles [78] detect mismatches between user interactions and previously defined developers expectations at runtime using software agents. Hilbert and Redmiles [79] discuss sequence comparison, i.e. the comparison of monitored event sequences with expected sequences. Furthermore, formal verification techniques can be applied to analyze user-system interactions before system deployment and identify potential problems and unexpected interactions (see e.g. [20, 21]). In contrast to MALTASE-based use case testing, these approaches do not target use cases.

A complementary approach for use case testing is to transform use cases into test cases and execute those to test software behavior. Several researchers, e.g. [29, 36, 72, 126, 144], proposed such an approach. In contrast, MALTASE-based use case testing compares use case steps directly to monitored user interactions without generating test cases.

Several researchers proposed approaches to reverse engineer use cases, i.e. to mine use cases from runtime traces. For example, El-Ramly et al. [45, 46] as well as Smit et al. [170] reverse engineer use cases from monitored interaction traces. Similarly, Antonio et al. [8] and Li et al. [107] reverse engineer use cases from code execution traces. These approaches assume that no use case documentation exists. In contrast, MALTASE-based use case testing assumes that a use case documentation written by developers exists and compares it to monitored user interactions.

Use cases are one type of software requirements and several researchers proposed approaches to monitor requirements at runtime. For example, Robinson discusses re-

quirements monitoring [147, 148] and monitors user goals at runtime [146]. Similarly, Feather et al. [48, 52] determine whether a running system meets its requirements by runtime monitoring and comparison to explicit specifications of requirements and developer assumptions. Furthermore, Wang et al. [197] present a framework for monitoring and diagnosing software requirements. MALTASE-based use case testing also monitores requirements but considers use cases instead of user goals.

5.4 Chapter Summary

This chapter presented three applications of the MALTASE framework: monitoring of user interactions preceding failures as reproduction steps to enable developers to reproduce failures (MALTASE-based failure reproduction), classification of user skills based on monitored user interactions to inform developers about skill levels of their users (MALTASE-based skill detection), and comparison of monitored user interactions with use case steps to detect wrong developer assumptions as indicators for software improvements and use case updates (MALTASE-based use case testing). The chapter motivated each framework application, described its implementation using components of the MALTASE framework, and discussed related work. These framework applications demonstrate how the MALTASE framework can be employed to acquire usage knowledge for developers. Furthermore, they illustrate how the same type of data - monitored, high-level user interactions - can be analyzed and exploited for different purposes.

Chapter 6

Evaluation

While the previous chapters described the MALTASE framework (Chapter 4) and its applications in software evolution (Chapter 5), this chapter describes an empirical evaluation of the MALTASE framework. Section 6.1 presents evaluation goals, research methods, and the target application used in the evaluation. The next three sections present results of three sub-evaluations targeting different aspects: Section 6.2 presents an evaluation of MALTASE monitoring by a simulation and a user survey. Section 6.3 presents the evaluation of MALTASE-based failure reproduction by a controlled experiment. And Section 6.4 presents an evaluation of MALTASE-based skill detection by an evaluation case study. Section 6.5 discusses conclusions and implications of all sub-evaluations. Finally, Section 6.6 summarizes the chapter.

6.1 Evaluation Overview

This section describes the "big picture" of the evaluation of MALTASE before subsequent sections provide more details and discuss its findings. Figure 6.1 illustrates on overview of the evaluation, more specifically the MALTASE components evaluated and the research methods used in the evaluation.

6.1.1 Goals

The purpose of the MALTASE framework is to monitor application usage and to extract actionable knowledge for developers during software evolution. The following aspects are crucial to investigate when evaluating the MALTASE framework: the performance overhead it introduces, whether users accept it, and its benefit and impact during software evolution.

Runtime monitoring introduces a performance overhead because it requires additional processing time and memory capacity. Hence, it is important to investigate this performance overhead and ensure that it does not hinder users in their work. We hypothesize that the performance overhead introduced by MALTASE monitoring does not hinder users in their daily work. We study this hypothesis in a simulation and a user survey which are presented in Section 6.2, investigating the performance overhead introduced by MALTASE monitoring in terms of execution time and memory consumption as well as the users opinion about it.

User acceptance, i.e. whether users agree to work with an instrumented application which monitors their interactions, is another important aspect besides technical

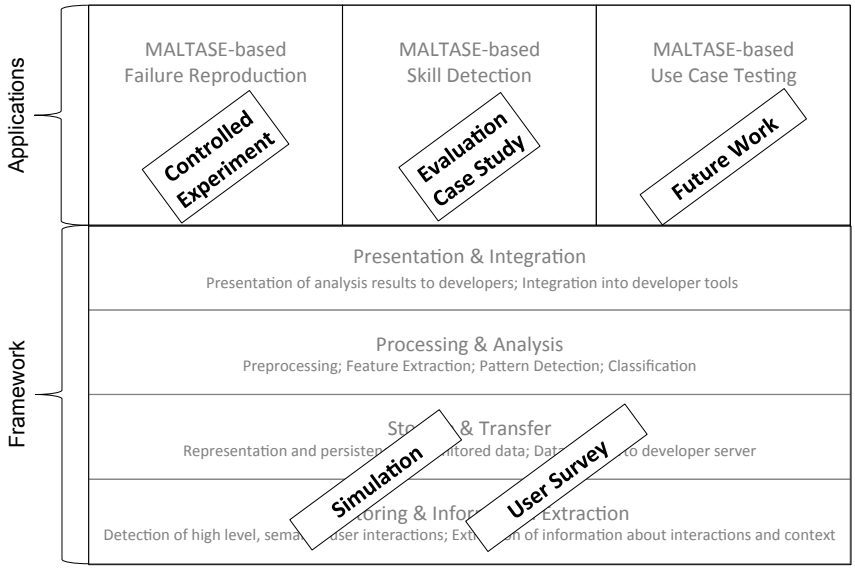


Figure 6.1: Overview of MALTASE Evaluation
Each box denotes a research method used in the evaluation

feasibility. We hypothesize that users accept MALTASE when the monitoring is performed anonymously and they perceive a benefit. We study this hypothesis in a user survey which is presented in Section 6.2, investigating whether users agree to work with a target application integrated with MALTASE, i.e. an application which monitors their interactions.

We evaluated two of the framework applications described in Chapter 5, MALTASE-based failure reproduction and MALTASE-based skill detection. MALTASE-based failure reproduction presents monitored user interactions preceding failures to developers during bug fixing. We hypothesize that developers can reproduce failures based on this information and that developers are enabled to reproduce failures they cannot reproduce before. We study this hypothesis in a controlled experiment which is presented in Section 6.3.

MALTASE-based skill detection detects user skills based on monitored user interactions. We hypothesize that user skills can be derived from user interactions monitored by the MALTASE framework. We study this hypothesis in an evaluation case study which is presented Section 6.4, investigating how well MOSKitt application skills and UML domain skills can be detected based on monitored user interactions.

6.1.2 Methodology

We conducted a simulation to measure the performance overhead introduced by MALTASE monitoring. We simulated user interactions and measured execution time and memory consumption for both an instrumented and plain version of the CASE tool MOSKitt. Comparing measurements for both versions allows to determine the time and memory overhead introduced by MALTASE monitoring. More details about the simulation are given in Section 6.2. The simulation provides "numerical

answers", but it cannot reveal whether the performance overhead hinders users in their work. For example, it is not clear from the simulation alone whether an increase in execution time of 10 % is acceptable for users or not.

To complement results of the simulation and investigate user opinions about MALTASE, we conducted a user survey. We deployed an instrumented version of the CASE tool MOSKitt to six MOSKitt users for two weeks. After this time, participants reported their opinion about performance overhead and privacy questions in a survey. This user survey is presented in Section 6.2.

To evaluate MALTASE-based failure reproduction, we monitored user interactions preceding failures and presented them to developers during bug fixing. We conducted a controlled experiment [42] to compare MALTASE-based failure reproduction to failure reproduction with textual bug reports submitted by users. The controlled experiment is presented in Section 6.3.

To evaluate MALTASE-based skill detection, we setup an evaluation case study to collect monitored user interactions as well as user skill levels for application and domain skill. This data collection was necesary because we did not find a dataset containing this information. We had 14 participants of varying skill levels perform three UML modeling tasks using MOSKitt, monitored their interactions, and collected self-estimations of MOSKitt and UML skill. This dataset was used to learn skill classifiers and evaluate their performance. The evaluation case study is presented in Section 6.3.

6.1.3 CASE Tool MOSKitt

The CASE tool MOSKitt[1] was used as target application during this evaluation. MOSKitt is an RCP-based open source desktop application for modeling and diagram-editing. MOSKitt supports modeling UML, BPMN, and entity relationship diagrams as well as capturing and managing software requirements in textual form. MOSKitt has been constantly developed since 2007 by 27 contributing developers and comprises 2 million lines of code which are mostly written in Java. Its user interface follows the WIMP style (Windows, Icons, Menus, Pointer) and consists of a project navigator, a palette for selecting model elements, and a diagram canvas (see Figure 6.2). We chose the UML (Unified Modeling Language) class diagram editor as context of the evaluation because it is the most prominent feature of MOSKitt and we had access to users of this feature. This editor allows users to create and modify UML class diagrams.

6.1.4 Integration of MALTASE Framework and MOSKitt

Figure 6.3 shows how the MALTASE components for monitoring, data transfer, and data persistence were deployed during the evaluation. MOSKitt is instrumented with the MALTASE *Sensors*. These sensors detect user interactions, extract context information about them, and pass this data via MALTASE *Monitoring Infrastructure* component the to the MALTASE *client*, which runs as a separate application on the user device. The MALTASE *Server Push-Client* component sends monitored date

[1] http://www.moskitt.org/ (Accessed Dec 2014)

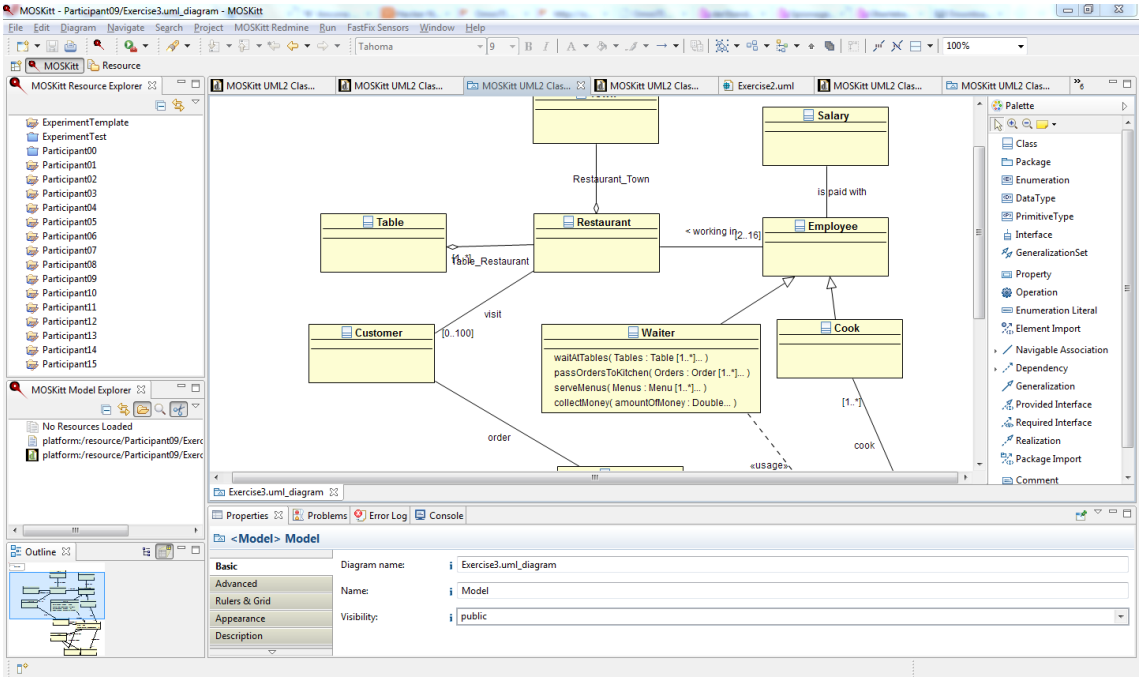


Figure 6.2: MOSKitt User Interface

via a TCP/ IP-channel to the developer server. The MALTASE *Server Push-Server* component receives data on the server side, passes them to the MALTASE *Database Data Store* which stores monitored data permanently in a database.

The deployment diagram shown in Figure 6.3 requires three setup steps. First, the MALTASE components have to be installed within MOSKitt using the RCP update mechanism. Second, the configuration (especially the TCP/IP settings of transfer components) have to be set appropriately to allow communication between client and server. And third, the MALTASE client and the developer server with the MALTASE server components have to be started.

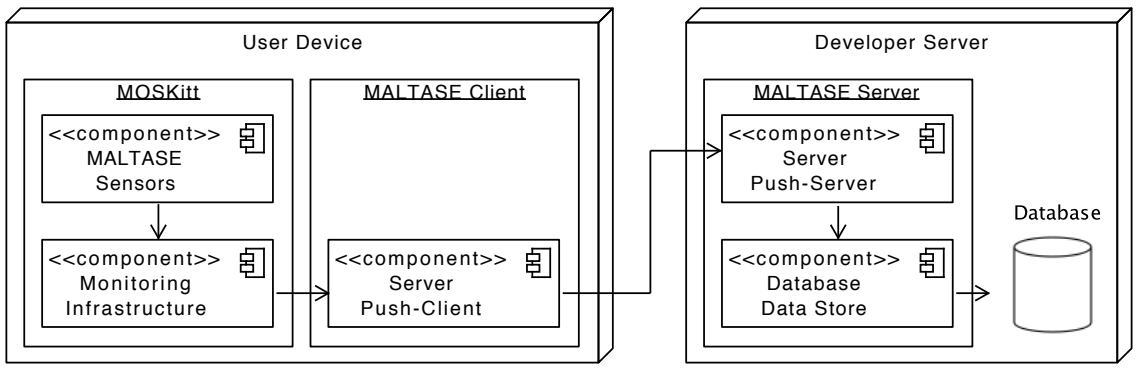


Figure 6.3: Deployment of MALTASE Components During Evaluation
Arrows denote data flow

6.2 Evaluation of MALTASE Monitoring

This section reports on the simulation and the user survey to study the performance overhead introduced by MALTASE monitoring and its the user acceptance. The evaluation was conducted in collaboration with an industry partner and its results have been partially published in [152]. This section describes design and results in more detail and presents unpublished results about privacy issues. Section 6.5 discusses conclusions and implications of the results.

6.2.1 Design

This evaluation uses two research methods, simulation and user survey, because they complement each other. We implemented the MALTASE components necessary to monitor user interactions, transmit them to a server, and store them in a database and integrated them with MOSKitt. The following MALTASE sensors were used: command sensor, menu and toolbar sensor, GUI part sensor, diagram manipulation sensor, application sensor, and exception sensor. We simulated user interactions, measured execution time and memory usage during the simulation, and compared those metrics of a plain MOSKitt instance with those of an instrumented MOSKitt instance. Six users of MOSKitt worked with an instrumented MOSKitt instance for two weeks and reported their feedback in a survey. Using both, simulation and user survey, was necessary because both methods complement each other: the simulation can measure performance overhead but cannot tell whether users perceive overhead as hindering their work. Similarly, the user survey can reveal the perception of users about privacy issues and performance overhead, but does not allow to quantify performance overhead.

Design of Simulation

We generated sequences of diagram manipulations and used them to simulate user interactions. This allowed to re-create the same user behavior multiple times and "apply" it to different MOSKitt instances. These sequences were simulated with a plain MOSKitt instance, i.e. a MOSKitt version without sensors, as well as an instrumented MOSKitt instance, i.e. a MOSKitt version with the MALTASE sensors. We measured and compared differences regarding time and memory consumption to evaluate the performance overhead introduced by the MALTASE sensors.

In this simulation, we consider diagram manipulations such as creating a diagram element as user interactions. The following types of diagram manipulations were simulated: Create/ Delete Project (CP, DP), Create Model (CM), Create/Update/Delete Element (CE, UE, DE), Execute Transformation (ET), Change Size (CS), Move Figure (MF), Switch Editor (SW), Undo (UN) and Redo (RE). The frequency of each manipulation type was determined randomly. As the order of manipulations is not arbitrary, e.g. an element can only be created when an umbrella model exists, a state machine was used to determine the next manipulation (see Figure 6.4). In MOSKitt, diagram elements are parts of models and models are organized in projects. The states InElement, InModel, and InProject represent the current focus of the user. During creation of the manipulation sequence, the

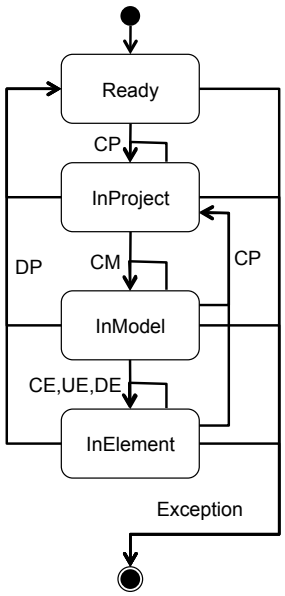


Figure 6.4: State Machine Used to Generate Sequences of Diagram Manipulations
C = Create, D = Delete, U = Update,
P = Project, M = Model, E = Element (Source: [152])

diagram is in a certain state and randomly selects a manipulation that is feasible in that state. If a manipulation is feasible in a state, it is represented as an edge leaving the state node. Projects can be created and deleted at any time and exceptions can occur at any time as well. Between two manipulations, a time break of 0, 1, or 2 seconds was inserted randomly.

The generated sequences of diagram manipulations were simulated with a plain MOSKitt instance and a MOSKitt instance instrumented with MALTASE sensors. Nine user sessions ranging in length from 50 to 800 diagram manipulations were simulated. The total time needed for the execution of all manipulations of a sequence was measured and the time per manipulation was calculated by dividing the total time by the number of manipulations. The average RAM consumption during a simulated session was calculated from periodical measures of RAM consumption. Those measures were collected using the nmon utility tool. A machine with a similar configuration as a typical user machine was used to run the simulation: A laptop equipped with a Intel Core 2 Duo processor (2 cores at 2.00 GHz) and 3 GB RAM which run Ubuntu Linux as operating system.

Design of User Survey

Six MOSKitt users (see Table 6.1 for details) worked with an MOSKitt instance instrumented with the MALTASE sensors for two weeks and reported their experiences in a survey. Participants were software developers working for the Spanish company which developed MOSKitt. They used MOSKitt in their software engineering work to create UML diagrams or document requirements textually. Participants worked with an instrumented version of MOSKitt for 5 days on average (min: 1 day, max: 8 days) and performed on average 6 sessions (min: 3 sessions, max: 12 sessions)

during this time. Two participants didn't use MOSKitt in their normal work while three participants used it daily. As this study was part of a larger study, the instrumentation consisted of the MALTASE sensors and additionally sensors for a record & replay approach that monitored application usage on a lower level of granularity. User feedback was collected using an anonymized, web-based questionnaire. Participants were asked the following questions:

- Do you agree with the following statement:
  "The application behaves the same way with as without the sensors." (Q1, agreement on a 5-item Likert scale Strongly Agree, Agree, Undecided, Disagree, Strongly Disagree)
- Did you notice changes in performance (e.g. longer response time) since you started using MOSKitt with sensors? (Q2, Yes/ No)
- If yes, do you agree with the following statement: "The performance overhead introduced by the sensors is tolerable and does not hinder my work." (Q3, agreement on a 5-item Likert scale Strongly Agree, Agree, Undecided, Disagree, Strongly Disagree).
- Do you agree that the following information about is sent to the maintenance team to help them fix errors quickly, given that the information is anonymous, i.e. it is not possible to establish my identity?
  ... information about my interactions (such as edits and deletions of diagram elements)
  ... information about my system (such as application version, Java version, Operating system version)
  ... text I enter in text fields
  ... the name of each file I manipulate
  ... the content of each file I manipulate
  (Q4, agreement on a 5-item Likert scale Strongly Agree, Agree, Undecided, Disagree, Strongly Disagree)

6.2.2 Results

Figure 6.5 shows the time and memory overhead between plain and instrumented MOSKitt instance during the simulations. The time overhead is 45 % for short sequences and converges to 5 % with increasing sequence length. We expected the overhead for short sequences as the sensor initialization overhead is proportionally large for sequences with few diagram manipulations. The average memory overhead introduced was 2-5 % independent of sequence length.

Table 6.1 summarizes the answers of survey participants. Five participants agreed or strongly agreed that they did not perceive a difference in the behavior of the instrumented MOSKitt compared with MOSKitt without sensors. Participant P2 disagreed and perceived a performance overhead, but judged it not to hinder his or her daily work. All six participants strongly agreed or agreed that user interactions, system information, and names of manipulated files can be monitored and sent to developers. For text entered and the content of manipulated files, the feedback of participants differed: while four participants agreed or strongly agreed that this information can be monitored, two participants disagreed or strongly disagreed.

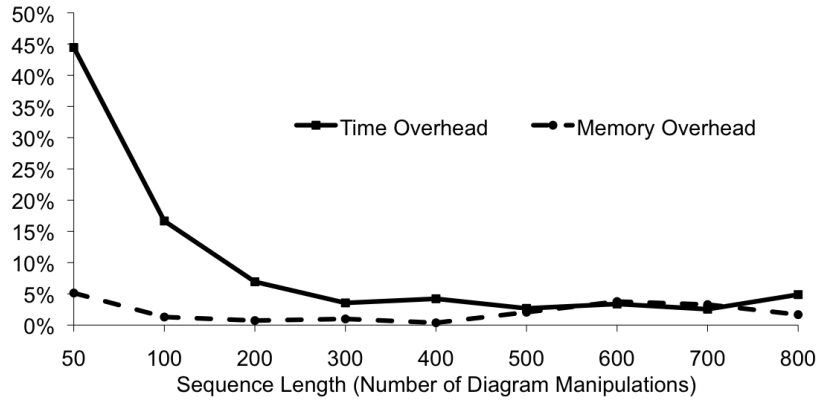


Figure 6.5: Performance Overhead Introduced by MALTASE Monitoring (Source: [152])

Table 6.1: Participants And Results of User Survey

Participant	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*
Information about Participants						
Days of Use	8	3	8	7	1	3
Sessions	8	3	12	7	4	3
Usual Usage Freq.	Never	Never	Daily	Daily	Daily	Weekly
Answers to Performance Overhead Questions (Q1-Q3)						
Same behavior (Q1)	Agree	Disagr.	Agree	Strong Agree	Agree	Agree
Perf. changes (Q2)	No	Yes	No	No	No	No
No hindrance (Q3)	-	Agree	-	-	-	-
Answers to Privacy Questions (Q4)						
User interactions	Strong Agree	Strong Agree	Strong Agree	Strong Agree	Agree	Strong Agree
System information	Strong Agree	Agree	Agree	Strong Agree	Agree	Strong Agree
Text entered	Strong Agree	Agree	Strong Agree	Strong Agree	Disagr.	Strong Disagr.
File names	Strong Agree	Agree	Strong Agree	Strong Agree	Agree	Agree
File content	Strong Agree	Agree	Strong Agree	Strong Agree	Strong Disagr.	Strong Disagr.

6.2.3 Limitations and Threats to Validity

We see the following four threats to validity for this evaluation. First, only diagram manipulations were simulated and therefore only the overhead of the diagram sensor as well as the processing and storage components are evaluated. As diagram manipulations are frequent user interactions in a diagram editor, we argue that the results are representative for diagram manipulation sessions. Second, generated sequences of diagram manipulations might deviate from real user behavior. To minimize this threat, the order of manipulations in generated sequences was determined randomly using the state machine and time breaks between two manipulations were introduced randomly. Third, an additional record & replay-instrumentation was running in parallel to the MALTASE sensors during the user survey. Therefore, it is not possible to determine the extent to which the MALTASE sensors are responsible for the performance overhead. But as this instrumentation introduced additional performance overhead and most users did not perceive any performance overhead, we argue that this fact does not affect the results. Fourth, the user survey was conduced with few participants in a cooperative setting, that is the participants knew the experimenters well and cooperated with them in a research project. Furthermore, participants were software developers themselves and probably do not represent the general user population. Hence, participants might be biased and future work should establish the generalizability of results regarding privacy.

6.3 Evaluation of MALTASE-based Failure Reproduction

This section reports on a controlled experiment evaluating MALTASE-based failure reproduction described in Section 5.1. The experiment was conducted in collaboration with Gurbanova [67] and published in Roehm et al. [152]. This section describes design and results of the experiment while Section 6.5 discusses conclusions and implications of the results.

6.3.1 Design

This controlled experiment was designed to evaluate MALTASE-based failure reproduction and compare it to failure reproduction with traditional, textual bug reports submitted by users. We used failures from bug reports in the MOSKitt bug repository in this experiment. To reflect the situation that many bug reports submitted by users do not contain reproduction steps [105, 203], bug reports were divided in two categories: bug reports that lack reproduction steps ($\text{BugReport}_{\text{MissingSteps}}$) and bug reports that contain reproduction steps ($\text{BugReport}_{\text{WithSteps}}$).

Experiment Setup

We conducted two sub-experiments. Experiment 1 studied whether developers can reproduce failures based on monitored user interactions preceding failures. Participants had to reproduce two bug reports from category $\text{BugReport}_{\text{WithSteps}}$. To avoid

a dependency of the results on the order of tasks, the order was chosen randomly. Members of the experimental group were given the Timeline Tool visualizing user interactions preceding the failure while members of the control group were given textual bug reports. Hence, the independent variable in experiment 1 was the representation of reproduction steps, either as visual interaction trace or in textual form. The dependent variable was whether participants could reproduce the failure. Experiment 2 studied whether MALTASE-based failure reproduction enables developers to reproduce failures which they cannot reproduce with textual bug reports. Each participant had to reproduce one bug report from category $\text{BugReport}_{\text{MissingSteps}}$. Members of the experimental group were given both the textual bug report and the Timeline Tool visualizing user interactions preceding the failure while members of the control group were given the textual bug report. Hence, the independent variable in experiment 2 was the availability of reproduction steps while the dependent variable was whether participants could reproduce the failure. While this seems to be "unfair" for members of the control group, it represents the usual situations of developers that bug reports submitted by users do not contain reproduction steps.

For each sub-experiment, we used a between subject design. We assigned participants to experimental group and control group randomly for experiment 1. After experiment 1, the groups of participants were switched, i.e. participants in the experimental group for experiment 1 were assigned to the control group in experiment 2 and vice versa. This design ensures that each participant works with the Timeline Tool. Table 6.2 gives an overview of the experiment setting.

Experiment Procedure

At the beginning of each session, participants were introduced to the UML modeling feature of MOSKitt and the Timeline Tool with a short video. Afterwards, each participant had to explore an exemplary interaction trace in the Timeline Tool and answer seven questions about it to make sure that they understood the usage of the tool. Then, experiment 1 and experiment 2 were conducted. Finally, participants filled a questionnaire to collect their opinion about the Timeline Tool and MALTASE-based failure reproduction.

During the experiment sessions, participants were observed and interesting observations were noted in a protocol. We logged hard results such as whether participants were able to reproduce a failure and how much time it took them. Additionally, we protocoled interesting participant behavior allowing to detect usability issues of the Timeline Tool, to identify extensions of the Timeline Tool, and to qualitatively analyze developer behavior during failure reproduction.

To help the experimenter conduct experiment sessions, we developed an experimenters guide (a step-by-step guide how to conduct an experiment session), an information sheet (a list of information pieces which should be given to participants at beginning of a session), an observer sheet (a template to record results and observations), and a questionnaire. This material was tested and revised with one participant and can be found in Appendix A.1.

Table 6.2: Experiment Setup

Failure	*Material for Experimental Group*	*Material for Control Group*
Experiment 1		
F1 (BR1)	Timeline Tool with user interaction trace	Textual bug report
F2 (BR2)	Timeline Tool with user interaction trace	Textual bug report
Experiment 2		
F3 (BR3)	Timeline Tool with user interaction trace, textual bug report	Textual bug report

Bug Report Selection and Trace Generation

To find suitable bug reports for the experiment, we analyzed the MOSKitt bug repository. Developers of MOSKitt reported that approx. 90 % of bug reports in the repository do not contain reproduction steps. The following inclusion criteria were used for selecting suitable bug reports: First, the failure reported in the bug report had to be reproducible in MOSKitt version 1.3.7 such that all of them could be investigated using the same MOSKitt instance. Second, the reported failure had to occur while using the UML modeling feature of MOSKitt to minimize familiarization effort for participants unfamiliar with MOSKitt. Third, the reported failure had to trigger an error dialog or an entry in the error log, ensuring that a failure reproduction can be detected clearly. Fourth, the bug report had to be in English. Using these criteria, four bug reports were selected.

These suitable bug reports were divided into the categories $\text{BugReport}_{\text{WithSteps}}$ and $\text{BugReports}_{\text{MissingSteps}}$. When we could reproduce a failure based on the description in the bug report, it was categorized as $\text{BugReport}_{\text{WithSteps}}$. In other cases, it was categorized as $\text{BugReport}_{\text{MissingSteps}}$ and a developer of MOSKitt was asked to provide reproduction steps. As one of the bug reports triggered several exceptions, it was used as example in the tutorial for the Timeline Tool and the other three bug reports BR1[2], BR2[3] and BR3[4] were used in the experiments. Table 6.3 provides details about these bug reports.

We simulated user interactions triggering each failure by performing the reproduction steps described in Table 6.3 in an instrumented version of MOSKitt. The MALTASE sensors recorded those simulated user interactions and created a user interaction traces which was shown to participants in the Timeline Tool. The following sensors were used: CommandSensor, GUI Part Sensor, Diagram Sensor, Exception Sensor, Application Sensor.

[2]https://moskitt.gva.es/redmine/issues/165 (Accessed Dec 2014)
[3]https://moskitt.gva.es/redmine/issues/138 (Accessed Dec 2014)
[4]https://moskitt.gva.es/redmine/issues/139 (Accessed Dec 2014)

Table 6.3: Bug Reports Used in Experiment

Id, Cat.	*Title: Description of Bug Report*	*Reproduction Steps*
BR1, With Steps	Error when assigning a StateMachine to a SubmachineState: When adding a "Submachine State", Moskitt throws an exception: Unhandled event loop exception java.lang.Stack OverflowError Then, the submachine is shown into the diagram, but it cannot be deleted.	1. Create new 'MOSKitt' project. 2. Create new UML2 diagram of type 'UML State machine' 3. Create a new 'Submachine State' element 4. Select the parent StateMachine as the referenced state machine -> An exception is thrown and diagram is no longer editable
BR2, With Steps	[Use Case] Error when create an extension point into a Use Case figure: [Use Case] Error when create an extension point into a Use Case figure	1. Create a new 'MOSKitt' project 2. Create new UML2 diagram of type 'UML UseCase' 3. Create a 'UseCase' element 4. Create an 'ExtensionPoint' inside use case -> An entry appears in the error log
BR3, No Steps	[Statemachine] Error when create a State Submachine: [Statemachine] Error when create an State Submachine	1. Create a new 'MOSKitt' project 2. Create new UML2 diagram of type 'UML State machine' 3. Create another UML2 diagram of type 'UML State machine' 4. Save both diagrams and keep them open 5. In the first diagram, create a 'Submachine State'. When prompted to select a State Machine, select the one from the other diagram. -> An exception is thrown and MOSKitt needs to be restarted

Table 6.4: Participants of Experiment
Position: S = Student, R = Researcher, D = Developer
Experience: Beg. = Beginner, Int. = Intermediate, Adv. = Advanced

Participant	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*
Position	S	S	S	S	S	S	S	S	R	R	D	D
Gender	M	M	M	M	F	F	M	F	F	M	M	M
MOSKitt Exp.	No	No	No	No	No	No	No	No	No	No	Yes	Yes
UML Exp.	Int.	Int.	Beg.	Beg.	Int.	Int.	Adv.	Int.	Int.	Adv.	Adv.	Adv.
Bug Fixing Freq.	Day	Week	Day	Day	Month	Never	Day	Day	Month	Week	Day	Month

Participants

Table 6.4 provides an overview of experiment participants. The experiment was conducted with 12 participants: eight master students, two researches, and two developers from the MOSKitt development team. Students and researchers did not have previous experience with MOSKitt. The self-assessment of UML experience, i.e. domain knowledge, ranged from beginner to advanced with mode intermediate (on scale Beginner - Intermediate - Advanced).

6.3.2 Results

This section presents both quantitative and qualitative results of the experiment. Conclusions from the results are discussed in Section 6.5. Table 6.5 provides an overview of quantitative results.

Quantitative Results

In experiment 1, six of six members of the experimental group could reproduce the failures based on interaction traces presented by the Timeline Tool. Similarly, six of six members of the control group could reproduce the failures based on textual bug reports. The average time needed by members of the experimental group was 3:30 min for F1 and 3:08 min for F2. Members of the control group needed 2:49 min and 2:05 min, respectively.
We conclude that developers can reproduce failures based on interaction traces presented by the Timeline Tool. As we did only a brief introduction of the Timeline Tool at the beginning of each experiment session, the additional time needed by members of experimental group might be explained with the time needed to familiarize with the Timeline Tool.

In experiment 2, four of the six members of the experimental group were able to reproduce the failure based on interaction traces presented by the Timeline Tool and textual bug reports. Two participants from the experimental group, P10 and P12, could not reproduce the failure. Participant P12 could not reproduce the failure because he was ignoring some interactions of the interaction trace. Participant P10 could not reproduce the failure because he was exploring the interaction trace backwards, i. e. right to left instead of left to right. In contrast, only one of six

Table 6.5: Quantitative Results of Experiment
Success: Was failure reproduced? (Yes/ No)
Time: Times taken in minutes
Group: E=Experimental Group, C=Control Group

Part.	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*
Experiment 1												
Group	C	E	E	C	C	E	C	E	E	C	E	C
Order	F1, F2	F1, F2	F1, F2	F2, F1	F1, F2	F2, F1	F1, F2	F1, F2	F2, F1	F2, F1	F2, F1	F1, F2
F1 Time	3:00	5:50	6:00	3:00	3:30	2:00	3:20	3:50	1:30	1:55	1:50	2:10
F1 Success	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
F2 Time	1:15	2:00	3:00	3:00	4:00	5:30	1:25	3:10	2:40	2:10	2:30	0:40
F2 Success	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Experiment 2												
Group	E	C	C	E	E	C	E	C	C	E	C	E
F3 Time	7:30	6:00	8:00	10:0	7:00	7:45	5:00	7:40	3:10	4:50	4:50	4:30
F3 Success	Y	N	N	Y	Y	N	Y	N	N	N	(Y)	N

members of the control group was able to reproduce the failure with a textual bug report. This was the developer that originally fixed the bug and he could reproduce it after a major hint from the experimenter. Members of the experimental group needed on average 6:28 min to reproduce the failure.
We conclude that MALTASE-based failure reproduction enables developers to reproduce failures when bug reports lack reproduction steps. This is a major improvement over the state of the practice, where reproduction steps are frequently missing in bug reports submitted by users.

Qualitative Results

While observing participants during the experiment, we identified two trace exploration strategies: Nine participants explored the trace chronologically, i.e. left to right, while three participants explored the trace backwards, i.e. right to left. The backwards strategy seems to resemble the analysis of a stack trace, where an exception is analyzed by exploring executed methods starting with the latest method (top down in the stack trace). To reproduce failures using the Timeline Tool, participants had to work chronologically but were not told this before. Also, they did not know that the Timeline Tool presents minimal reproduction traces, i.e. all user interactions of an interaction trace have to be performed to reproduce a failure. We hypothesize that participants who explored the trace from right to left tried to identify the first interaction necessary for failure reproduction.

All participants provided with both bug report and Timeline Tool analyzed the bug report first. When they noticed that the textual bug report did not contain reproduction steps, all of them switched to the Timeline Tool. We hypothesize that

participants refer to textual bug reports first because they are well known. Also, textual bug reports can give a quick impression about the failure. One participant proposed to integrate both, the interaction trace and the textual bug report, in a common view to facilitate the exploitation of both data sources simultaneously. We agree and think that this is an interesting direction for future work.

We collected feedback of participants about MALTASE-based failure reproduction and its impact on failure reproduction in a short questionnaire. Eleven participants agreed (8) or strongly agreed (3) that the meaning of information presented in the Timeline Tool is clear and easy to understand. Similarly, eleven participants agreed (7) or strongly agreed (4) that it is easy to find required information in the Timeline Tool during failure reproduction. Eleven participants agreed (3) or strongly agreed (8) that the Timeline Tool is helpful when reproducing failures. Similarly, eleven participants agreed (6) or strongly agreed (5) that it is clear what the user did by analyzing the interaction trace in the Timeline Tool. Eight participants preferred the interaction traces as presented by the Timeline Tool while two participants preferred textual bug reports, given that both contain the same information.

6.3.3 Limitations and Threats to Validity

As every empirical study, the design of this experiment has threats to validity. Interaction traces presented to participants in the Timeline Tool were generated manually. Thereby, we followed reproduction steps elicited from the textual bug reports or provided by MOSKitt developers. This procedure ensures that the traces consist of real user interactions. It allows to test whether developers comprehend user interactions presented in the Timeline Tool and whether they can reproduce failures based on this information. But it has the limitation that traces contain no noise, i.e. user interactions not necessary for failure reproduction. Hence, future work could investigate failure reproduction with interaction traces containing noise as well as approaches to automatically extract reproduction steps from user interaction traces. Furthermore, each participant reproduced three bug reports for one application. Because real bug reports and a real world application were used in the experiment, we argue that the experiment represented developers' situation in the real world. But future work should study external validity, i.e. to what degree experiment results generalize to other types of failures and other applications. Also, participants reproduced failures on a laptop with the same MOSKitt instance which was used to generate the interaction traces. Hence, the impact of MALTASE-based failure reproduction for changing environments should be investigated and it should be combined with approaches targeting the reproduction of failure environments.

6.4 Evaluation of MALTASE-based Skill Detection

This section reports on an evaluation case study to evaluate MALTASE-based skill detection which is described in Section 5.2. The data collection was conducted in collaboration with Theiner [184]. This section presents design and results of the case study while Section 6.5 discusses conclusions and implications of the results.

6.4.1 Design

This evaluation case study is a confirmatory case study [42] investigating the hypothesis that user skills can be inferred from user interactions. We used the UML editor MOSKitt as target application and consider two types of user skill following the skill categories of Nielsen [127]: MOSKitt application skill, i.e. skill with regard to the MOSKitt application, and UML domain skill, i.e. skill regarding modeling UML class diagrams. As we could not find a dataset containing user interactions as well as MOSKitt and UML skill levels, we collected the data ourselves. We had study participants work on UML modeling tasks in MOSKitt and captured their interactions during those task with the MALTASE sensors. Furthermore, we asked participants for self-estimations of their skill. This dataset was used to learn and evaluate skill classifiers.

Tasks

Each participant had to complete the same three tasks which were UML modeling exercises from the book by Bruegge and Dutoit [23]. The level of difficulty increased with each task: While the first two tasks were closed tasks, i.e. their solution was obvious from the task description, the last task was open, i.e. the solution was not obvious from the task description and different solutions were possible.

The following three tasks were used during data collection of the study (the full task descriptions are given in Appendix A.2):
Task T1: Participants were given a UML class diagram on a sheet of paper and asked to create the same class diagram in MOSKitt. The diagram consisted of three classes and three associations.
Task T2: Participants were given an existing UML class diagram in MOSKitt which consisted of six classes but no associations. Each class represented a geometric figure such as a polygon or a rectangle. Additionally, they were given a textual description of associations between those classes, e.g. "a group consists of figures" or "a polygon consists of at least three lines". The task was to add six associations described in the text to the existing class diagram.
Task T3: Participants were given the following textual description of a "restaurant world": "Create a UML model for the following situation: In a small town there are five restaurants with a number of employees ranging from two in the smallest to 16 employees in the biggest restaurant. Each employee has an individual salary. There are two types of employees: cooks and waiters. Waiters wait on tables, pass the orders to the kitchen, serve the menus and collect the money from the customers. The cooks prepare the menus. Each menu has its own price depending on the dishes, but each consists of at least a salad and a main course. Customers visit restaurants and order menus. The largest restaurant can serve up to 100 customers at a time, at bad times a restaurant might have no customers at all." Participants had to model this "restaurant world" as a UML class diagram.

Participants

The study design requires a basic understanding of UML class diagrams. Hence, we recruited participants with at least basic UML class diagram skills. To detect different skill levels regarding MOSKitt and UML, we aimed to recruit participants with different skill levels, i.e. skill combinations like "UML beginner, MOSKitt beginner", "UML advanced, MOSKitt beginner", or "UML advanced, MOSKitt advanced". The combination "UML beginner, MOSKitt advanced" was excluded because as experience with MOSKitt usually requires also UML experience as MOSKitt is mainly used as a UML tool. Three groups of people represent these skill combinations: students with basic UML skills from software engineering classes but no experience with MOSKitt; software engineering researchers with UML skills but no experience with MOSKitt, and finally software developers from the MOSKitt team with both UML and MOSKitt skills. Hence, we recruited participants from these groups.

We conducted the experiment with 15 participants: six students, seven software developers, and two researchers. Table 6.6 provides an overview of the participants. Five developers were members of the MOSKitt development team. We collected self-estimations of participants for MOSKitt skill and UML skill on a four-item scale (None, Beginner, Intermediate, Advanced). Ten participants had not worked with MOSKitt before, one judged him- or herself on intermediate level, and four on advanced level. Six participants judged themselves to be on beginner level regarding UML class diagram modeling, five on intermediate level, and four on advanced level. Because skill self-estimations may not be reliable, we additionally determined skill levels independently of participant's self-estimations: We classified MOSKitt skill of participants based on their MOSKitt experience on a two-item scale as novices or experts. Furthermore, we classified UML skill of participants based on their solution to task T3 on a three-item scale as beginner, intermediate, or advanced. All participants had a high skill level regarding computer usage as they frequently use computers during their daily work. Experiment sessions of six participants were conducted remotely via screen sharing while the sessions with the other eight participants were conducted on site. Remote participation was necessary because the MOSKitt development team was not co-located. For each participant, the time needed for each task and the complexity of the solution for task T3, i.e. the number of UML elements in the solution, was recorded and is given in Table 6.6.

Research Process

Figure 6.6 provides an overview how the study was the dataset was collected and how skill classifiers were learned and evaluated from this dataset. In the following we briefly describe each step.

Data Collection Phase All study sessions were conducted by one experimenter with one participant. If a participant had not used MOSKitt before, a short demo video explaining the creation of UML class diagrams in MOSKitt was shown at the beginning of a session. Participants were provided with a laptop on which an instrumented MOSKitt instance was running as well as a task sheet describing the three tasks. Each participant had to solve the tasks consecutively without addi-

Table 6.6: Overview of Case Study Participants
'Remote?' denotes participation via screen sharing
'Team?' identifies members of MOSKitt development team

Id	*Sex*	*Occupation*	*MOSKitt Skill (Rep.)*	*MOSKitt Skill (Aggr.)*	*UML Skill (Rep.)*	*UML Skill (Metric)*	*Remote?*	*Team?*	*Time T1 in min*	*Time T2 in min*	*Time T3 in min*	*Compl. T3 Soln.*
P1	M	Stud.	None	Novice	Interm.	Beg.	No	No	10.8	9.4	8.3	21
P2	M	Dev.	Interm.	Expert	Interm.	Adv.	Yes	Yes	3.7	5.7	24.5	26
P3	M	Dev.	Adv.	Expert	Interm.	Interm.	Yes	Yes	6.8	2.9	9.9	29
P4	M	Stud.	None	Novice	Interm.	Interm.	No	No	7.7	6.3	27.1	26
P5	M	Res.	None	Novice	Adv.	Adv.	No	No	6.9	4.8	10.8	No
P6	N	Dev.	Adv.	Expert	Interm.	Adv.	Yes	Yes	11.1	5.1	10	30
P7	F	Dev.	Adv.	Expert	Interm.	Interm.	Yes	Yes	5.9	3.8	21.2	34
P8	F	Dev.	Adv.	Expert	Adv.	Adv.	Yes	Yes	7.0	4.4	26.3	57
P9	M	Stud.	None	Novice	Interm.	Interm.	No	No	6.3	5.0	9.7	24
P10	M	Stud.	None	Novice	Beg.	Interm.	No	No	5.5	3.9	11.8	26
P11	M	Stud.	None	Novice	Beg.	Beg.	Yes	No	6.0	12.8	20.6	26
P12	M	Dev.	None	Novice	Beg.	Beg.	No	No	12.7	11.9	32	26
P13	M	Dev.	None	Novice	Interm.	Interm.	No	No	12.8	5.0	22.3	40
P14	M	Stud.	None	Novice	Interm.	Interm.	No	No	4.3	7.9	9.4	21

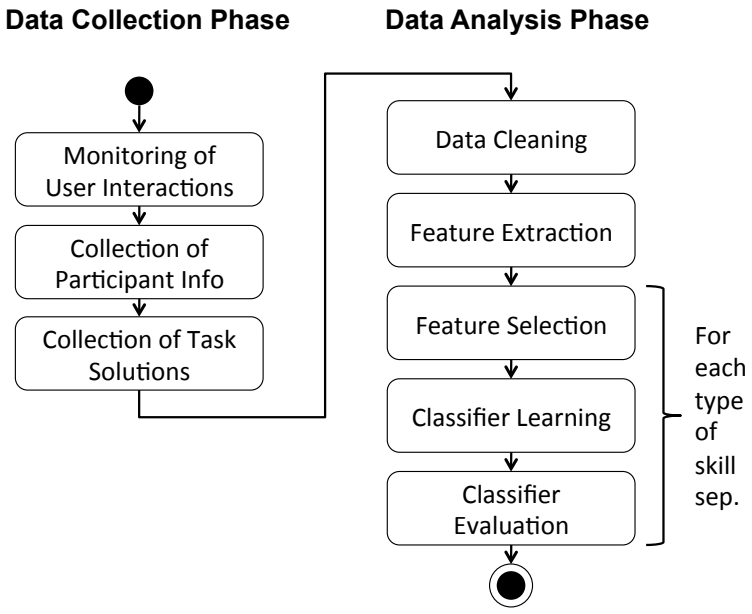


Figure 6.6: Research Process of Evaluation Case Study

tional help. Participants could work on each task until they felt to be done. While participants worked on the tasks, their interactions were recorded with the RCP Framework Sensors of the MALTASE framework (step 'Monitoring of user interactions' in Figure 6.6). The following MALTASE sensors were used: menu and toolbar sensor, GUI part sensor, diagram sensor, and command sensor. Furthermore, we implemented additional sensors which monitor mouse actions like mouse clicks and mouse movements as well as hot keys. At the end of a session, the participant had to complete a questionnaire which collected self-estimations of MOSKitt UML Class Editor skill and UML Class Diagram skill on a four level scale (Never used, Beginner, Intermediate, and Advanced) together with general participant information (step 'Collection of Participant Info' in Figure 6.6). Furthermore, task solutions were saved for later analysis (step 'Collection of Task Solutions' in Figure 6.6). The experimenter observed participants during their work and recorded observations.
To help the researcher conducting study sessions, we designed a study guide (a step-by-step guide how to conduct a session), an information sheet (a list of information pieces which should be given to participants at beginning of a session), an observer sheet (a template to note results and observations), and a questionnaire. This material was tested in two test runs and revised after each run. It can be found in Appendix A.2. Six sessions were conducted remotely via screen sharing because the corresponding participants were not co-located. In these cases, participants took control over the study laptop via screen sharing and remotely interacted with MOSKitt. The task sheet was presented next to MOSKitt on the screen.

Data Cleaning The dataset collected during study sessions was cleaned and extended in several ways which are described in the following.
To control for usability issues due to screen sharing, we asked remote participants if they experienced usability problems during their session. This was the case for one participant and we removed his or her data from the dataset before classifier learning and evaluation (this participant is not shown in Table 6.6).
Because skill self-estimations are typically not reliable, we aggregated the self-reported MOSKitt skill estimations (column 'MOSKitt Class Diagram Editor Skill (Reported)' in Table 6.6) to improve the quality of our data. Nine participants had never worked with MOSKitt before the study. Therefore we categorized them as 'novice' (column 'MOSKitt Class Diagram Editor Skill (Aggregated)' in Table 6.6). Five participants from the MOSKitt development team judged their MOSKitt UML class editor skill as 'intermediate' or 'advanced'. Consequently, we categorized them as 'experts'. We validated this categorization by running the k-means clustering algorithm with k=2 clusters on the dataset. All but one 'expert' participants were assigned to one cluster and all 'novice' participants were assigned to the other cluster. Hence, we are confident that the aggregated skill levels are reasonable. In case of UML class diagram skill, no such clear distinction could be made because all participants had experience with UML class diagrams and skill self-estimations ranged from 'beginner' to 'advanced' (column 'UML Class Diagram Skill (Reported)' in Table 6.6). Therefore we analyzed participants' solutions to task T3 in order to validate UML skill self-estimations. We investigated differences between correct, complete solutions and incorrect, incomplete solutions. We found three dimensions where

solutions differed: frequent use of advanced modeling constructs (namely datatypes and unbounded multiplicities such as '1..*'), the number of modeling errors, and the fraction of information given in the task description but not modeled in the solution. We designed an expert and a novice predictor for each of these three dimensions separately: Participants were classified as experts if they used each advanced modeling constructs more than once while they were classified as novices if they used each construct at most once. Further, participants were classified as experts if they made at most one error in their T3 solutions, while they were classified as novices if they made more than three errors. Finally, participants were classified as experts when they modeled more than 90% of the information contained in the task description, while they were classified as novices when they modeled less than 66%. The parameter values of these predictors were determined by manual analysis and comparison of T3 solutions. A majority vote of these predictors was used to determine the UML class diagram skill of each participant (column 'UML Class Diagram Skill (Metric)' in Table 6.6). If participants were estimated neither as 'expert' or 'novice' in the majority vote, they were categorized as 'intermediate'. We also ran k-means clustering with k=3 to validate the UML class diagram skill levels. But we could not find an assignment of participants to clusters which resembled the UML skill distribution, neither for the self-reported skill estimations nor for the metric.

Feature Extraction Classification features such as the number of command actions summarize a user session and are needed by skill classifiers as input. They are extracted from monitored interaction traces by the feature extractor component. Table 6.7 provides an overview of the classification features used. We extracted 88 classification features in seven categories and analyzed which feature subset performs best to predict user skill. We extracted seven different types of features: general features, features based on mouse actions, features based on keyboard actions, features based on interactions with the WIMP user interface, features specific for MOSKitt, features specific for UML modeling, and task specific features. For each feature type, the absolute count as well as the frequency per minute were extracted. For example, the total number of actions during a study session as well as the number of actions per minute were used as a classification feature. We motivate some features in the following: The feature "palette hover actions" (F11) resembles that MOSKitt users have to select a UML modeling construct such as a class or an inheritance relation from a tool palette before they can then add it to the diagram canvas. Hence, we hypothesize that hovering over a tool palette item indicates exploration. Further, the features "add-delete-add pattern?" (F12) and "add-delete-add pattern" (F13) resemble a usability problem which influenced many novice MOSKitt users: when drawing an association between two UML classes, the direction was counterintuitive and many novice users had to delete the wrong-directed association and recreate it with the right direction. Task-dependent features were manually derived from interaction traces or the solution of task T3.

Feature Selection Extracted feature values were used to learn skill classifiers. We used decision trees as classifiers for user skill. Ideally, a decision tree learning algorithm is able to automatically choose the optimal subset of features for con-

Table 6.7: Classification Features Extracted From Interaction Traces
'Runtime?' denotes if a feature can be extracted at runtime

Id	*Shorthand*	*Description*	*Runtime?*
General Features			
F1	#actions	Number of (all types of) actions	Yes
Features Based on Mouse Actions			
F2	#clicks	Number of mouse clicks	Yes
F3	#mouse pauses	Number of pauses between two mouse actions	Yes
F4	Øpause duration	Average pause duration between two mouse actions	Yes
Features Based on Keyboard Actions			
F5	hot keys used?	Indicates whether hot keys were used	Yes
F6	#hot key actions	Number of hot key actions	Yes
Features Based on Interactions With WIMP User Interface			
F7	#commands	Number of command actions	Yes
F8	#menu actions	Number of main menu and context menu actions	Yes
F9	#tool actions	Number of toolbar actions	Yes
F10	#UI part switches	Number of switches to another part of the UI	Yes
Features Based on Target Application "MOSKitt"			
F11	#palette hover actions	Number of hovers over tool palette items (tool palette is MOSKitt specific UI-component to select UML modeling elements)	Yes
F12	add-delete-add pattern?	Indicates whether interaction pattern "Add assoc., delete it, add it again" occurred	Yes
F13	#add-delete-add pattern	Number of occurrences of pattern "Add assoc., delete it, add it again"	Yes
Features Based on Target Domain "UML Class Diagram Modeling"			
F14-F33	diagram manipulation actions	Number of actions which add or remove a particular type of UML class diagram modeling construct, e.g. AddUmlClass, AddUmlOperation, RemoveUmlAssociation, RemoveUmlProperty	Yes
Features Based on Task			
F34	task time	Time needed to complete each task, i.e. T1, T2, T3	No
F35	solution complexity T3	Complexity of solution for task T3; measured as the total number of classes, relations, operations, attributes, and multiplicities	No

structing the decision tree. But it has been shown that even random features can decrease classifier performance because of the greediness of decision tree learning [199]. Hence, a feature selection step is necessary to identify the optimal feature subset for learning decision tree classifiers. We selected features for decision tree learning in two sub-steps. First, we used a filter method [199] to determine a first subset of features: All features were ranked according to their information gain with respect to the currently investigated skill type and all features with a low information gain were dropped. Second, the first feature subset was further reduced by a wrapper method [199]: A decision tree classifier was learned for all feature combinations of the first feature subset and evaluated against a training set. The combination of features with the best result was used for classifier learning. This analysis was done individually for each skill type using the weka data mining toolkit [69].

Classifier Learning We learned decision tree classifiers for the following four types of user skill (see Table 6.8): self-estimated MOSKitt UML class diagram editor skill (four-item scale: Never used, Beginner, Intermediate, Advanced), aggregated MOSKitt UML class diagram editor skill (two-item scale: Novice, Expert), reported UML class diagram skill (three-item scale: Beginner, Intermediate, Advanced), and UML class diagram skill derived using the metric described above (same three item scale). A decision tree classifier was learned for each type of skill separately using the feature subset identified during feature selection. Decision trees were learned using weka's J48 algorithm [69] which is an implementation of the C4.5 decision tree learning algorithm [141]. We chose decision trees since they are well-known classifiers and lead to human-interpretable classifiers. We also experimented with different feature subsets when learning skill classifiers: all features (all features described in Table 6.7), independent features (all features independent of application, domain, and task), runtime features (features which can be derived online at runtime), and UML modeling features (features derived from UML modeling actions).

Classifier Evaluation Each skill classifier was evaluated using 10-fold cross validation. The results of the classifier evaluation are summarized in Table 6.8.

6.4.2 Results

This section presents results from classifier learning and evaluation as well as observations made during study sessions. Conclusions and implications from these results are discussed in Section 6.5.

Results From Classifier Learning and Evaluation

We learned and evaluated classifiers for different types of user skill, namely MOSKitt Skill (Reported), MOSKitt Skill (Aggregated), UML Skill (Reported), and UML Skill (Metric). Table 6.8 provides an overview of the results obtained. 'Accuracy' denotes the fraction of participants which were classified correctly. The kappa coefficient κ is a chance-corrected measure that expresses the agreement between classified

Table 6.8: Results of Classifier Learning and Evaluation

Skill Type	*Features Used*	*Accuracy*	κ	*Prec. (Wgt. Avg.)*	*Recall (Wgt. Avg.)*	*Tree Size*	*Classification Features in Decision Tree*
Application Skill							
MOSKitt Skill (Rep.)	All	86%	0.69	0.79	0.86	3	EdgeRemoved (F14-F33)
	Runtime	86%	0.69	0.79	0.86	3	EdgeRemoved (F14-F33)
MOSKitt Skill (Aggr.)	All	93%	0.84	0.94	0.93	3	EdgeRemoved (F14-F33)
	Runtime	93%	0.84	0.94	0.93	3	EdgeRemoved (F14-F33)
Domain Skill							
UML Skill (Rep.)	All	64%	0.26	0.59	0.64	7	AddDeleteAddPattern (F13), GuiPartActivation (F10)
	Runtime	64%	0.26	0.59	0.64	7	AddDeleteAddPattern (F12), GuiPartActivation (F10)
UML Skill (Metric)	All	71%	0.51	0.76	0.71	7	AddDeleteAddPattern (F13), PaletteHover (F11)
	Runtime	71%	0.51	0.76	0.71	7	AddDeleteAddPattern (F12), PaletteHover (F11)

skill levels and true skill levels [199]. Its value indicates the performance improvement over a random classifier. Precision and recall values for each classifier are given averaged, i.e. the number of participants on each skill level are taking into account when calculating these metrics.

The accuracy of classifiers for the reported MOSKitt skill was 86%. Their kappa value κ was 0.69, indicating a better performance than a random classifier. The classifiers also outperform a ZeroR [199] classifier (a classifier which always predicts the most common value of the target skill level) with an expected accuracy of 64 %. All decision trees consist of only three nodes, indicating that few features suffice to classify user skill which translates to a low performance overhead as only these few features have to be monitored and extracted. Moreover, the classifier learned from runtime features only performs as well as the classifier learned from all features. This observation indicates that runtime information suffices to detect user skill and that skill detection can be employed during runtime, i.e. while the user is interaction with an application. When ignoring features related to the MOSKitt usability issue (i.e. features EdgeRemoved (F14-F33) and add-delete-add pattern (F12, F13)) during classifier learning, the classifier performance slightly decreases to 79% accuracy (κ: 0.53, precision: 0.73, recall: 0.79).
Figure 6.7 shows the decision tree learned for aggregated MOSKitt skill from all features. It consists of one test of the feature EdgeRemoved (F14-F33), i.e. how often an edge in a UML class diagram was removed during a session. When a user

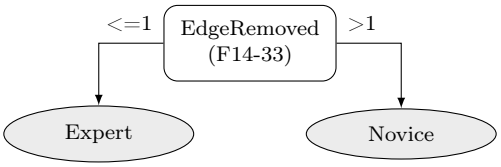


Figure 6.7: Decision Tree for MOSKitt Skill (Aggr.) Learned From All Features

removed at most one edge during a session, he or she is classified as an expert user. Otherwise the user is classified as a novice users.

Classifiers for aggregated MOSKitt skill performed slightly better with an accuracy of 93%. Their kappa value κ was 0.94, indicating that the classifiers clearly outperform a random classifier. Similarly, the classifiers outperform a ZeroR classifier with expected accuracy of 64%. We observed an equal performance of both classifiers learned from all features and classifiers learned from runtime features only. Similar to classifiers for reported MOSKitt skill, the classifier learned from all features and the classifier learned from runtime features only show the same performance. Furthermore, the size of the decision trees is small. When ignoring features related to the MOSKitt usability issue (i.e. features EdgeRemoved (F14-F33) and add-delete-add pattern (F12, F13)) during classifier learning, the classifier performance decreases to 86% accuracy (κ: 0.69, precision: 0.86, recall: 0.86).

The accuracy of classifiers for reported UML skill was 64%. The kappa value κ of 0.26 indicates a small performance improvement compared to a random classifier. Compared to a ZeroR classifier with an expected accuracy of 64%, learned classifiers show the same performance. Similar to classifiers for reported MOSKitt skill, the classifier learned from all features and the classifier learned from runtime features only show the same performance. Furthermore, the size of the decision trees is small. When ignoring features related to the MOSKitt usability issue (i.e. features EdgeRemoved (F14-F33) and add-delete-add pattern (F12, F13)) during classifier learning, the classifier performance remains similar with 64% accuracy (κ: 0.3, precision: 0.59, recall: 0.64).

Classifiers for UML skill derived using the metric perform performed slightly better with accuracy of 71%. They perform better than a random classifier (kappa value κ of 0.51) and a ZeroR classifier with expected accuracy of 50 %. Similar to classifiers for reported MOSKitt skill, the classifier learned from all features and the classifier learned from runtime features only show the same performance. Furthermore, the size of the decision trees is small. When ignoring features related to the MOSKitt usability issue (i.e. features EdgeRemoved (F14-F33) and add-delete-add pattern (F12, F13)) during classifier learning, the classifier performance decreases to 57% accuracy (κ: 0.23, precision: 0.45, recall: 0.57).

Observations During Study Sessions

We made several observations during the study sessions which provide insights about skill classification and classification features. Users with a high skill level completed the easy tasks T1 and T2 in less time than less skilled users. We made this observation for both application and domain skill. Interestingly, this observation was

not true for the more complex task T3. Here, several participants with high skill level took much longer time because they modeled the solution in much more detail. Hence, task completion time should not be used as an indicator for user skill alone.

We identified a usability issue in MOSKitt during the study which affected many novice MOSKitt users when creating directed associations between two classes. To accomplish this task, a user has to choose an association type from the tool palette, click on the source class, keep the mouse pressed, move the cursor to the destination class, and release the mouse. Several novice MOSKitt users released the mouse too early or selected the classes in the wrong order. When users faced this situation, they had to delete the association and create it again correctly. We call this behavior the "Add association, Delete association, Add association" pattern and hypothesize that it is indicator for low MOSKitt skill (cf. features F12 and F13 in Table 6.7).

Furthermore, we found that frequency of hot keys was inversely proportional to user skill level. This was to our surprise as we had assumed that hot key usage is an indicator for high user skill. When examining this observation in more detail, we found that 'F2' hot key was the most frequently used hot key. This hot key is used to rename a model element in MOSKitt. When a new model element is created in MOSKitt, it is assigned a default name. The user can immediately change the default name by overwriting the default name right after the creation of the model element. While experienced MOSKitt users renamed model elements at the time of creation, this option was not obvious for new MOSKitt users. Hence, new MOSKitt users used the F2 shortcut to change element names. We conclude that a particular classification feature can be an indicator for high user skill in one context while it can be an indicator for low user skill in another context. This requires a careful analysis of classification features when transferring skill classifiers from one context to another one, e.g. from one application to another application or from one domain to another domain.

6.4.3 Limitations and Threats to Validity

As every empirical study, the design of this case study has limitations and threats to validity. The case study was conducted with a single application, MOSKitt, within a single domain, UML class diagram modeling. Future work has to establish the generalizability of the results to other applications and domains.
Furthermore, 14 participants took part in the study. Because of this medium number of participants, results should be interpreted carefully and conservatively. We argue that this number is big enough to obtain insights but it remains future work to establish the generalizability of the results. However, many comparable studies were conducted with a similar number of participants.

Study participants self-estimated their skill levels regarding MOSKitt and UML class diagram modeling. While this information is easy to obtain, it might not always correspond to the true skill level or be inconsistent among different participants. To address this threat, we determined participants' skill levels independently of their own ratings as described above.

User skills change over time when users become experienced with an application or a task domain. Moreover, different types of user skill might be correlated with each other. These two characteristics of user skill were not considered in this study.

During the study, a usability problem of MOSKitt was detected which mainly affected novice MOSKitt users and forced them to correct their UML diagram by removing and re-creating associations between classes. The features EdgeRemoved (F14-F33) and add-delete-add pattern (F12, F13) reflect this behavior and those features were used in several classifiers to discriminate between novice and expert MOSKitt users. Hence, it is not possible to assess whether these classifiers are able to discriminate between novice and expert users in the absence of this usability problem. But the observation that MOSKitt skill can be reliably predicted without these features shows that there are alternatives to using those features for classification.

We conducted seven study sessions remotely via screen sharing because all advanced MOSKitt users were located abroad. We asked all remote participants whether they faced usability problems due to screen sharing and did not use their data if this was the case. Several remote participants commented that they could not use all hot keys they were used to, which might influence results regarding hot key usage. But we observed that work on UML modeling tasks in MOSKitt requires mainly interactions with the tool palette and diagram canvas using the mouse. Moreover, all tasks could be completed without using hot keys.

6.5 Discussion

The evaluation found that MALTASE monitoring introduces a time overhead of 5% and a memory overhead of 2-5%. Users do not perceive this performance overhead or judge it as not hindering their work. Hence, we conclude that MALTASE monitoring is feasible without introducing performance problems for users.

Survey participants agreed that their interactions with the application as well as system information is monitored and collected anonymously. Hence, we conclude that MALTASE can be used in the context represented by study participants, i.e. software developers working with CASE tools. Future work should investigate whether this result holds for users in general. Related work by Sheth et al. [166] found that interaction data is considered less sensitive than document content or personal data by users.

We did not directly evaluate the size of interaction traces and whether data of many users can be stored and transferred by components of the Data Storage & Transfer layer. But as no problems occurred during the user survey study (where six MOSKitt users worked with an instrumented MOSKitt instance connected to one developer server), we assume that the MALTASE framework can handle interaction traces from a moderate number of users. Future work could investigate how MALTASE handles interaction traces of a large user base and whether load balancing mechanisms are necessary in that case.

Participants working with MALTASE-based failure reproduction could reproduce a failure in 16 of 18 instances. Hence, we conclude that developers can reproduce fail-

ures based on monitored user interactions preceding failures. Further, we conclude that the MALTASE sensors provide enough information for failure reproduction.

The evaluation found that participants unfamiliar with MOSKitt were able to reproduce failures they could not reproduce with textual bug reports only: The four students working with textual bug reports could not reproduce the failure in experiment 2 while the four students working with MALTASE-based failure reproduction could reproduce it. As the students were master students and most of them worked as part-time developers, we hypothesize that they mimic developers without MOSKitt experience. Hence, we conclude that MALTASE-based failure reproduction enables developers unfamiliar with an application to reproduce failures they cannot reproduce with bug reports because of missing reproduction steps in the reports. This is a major improvement over the state of the practice where bug reports submitted by users often do not contain reproduction steps [105, 203].
As only two developers from the MOSKitt team participated in the evaluation, we refrain from making conclusions about the impact of MALTASE-based failure reproduction for developers experienced with an application. But the positive feedback from both developers about MALTASE-based failure reproduction indicates an interest and impact for developers, too.

The interaction traces used in the evaluation did not contain noise, i.e. interactions which are not necessary for failure reproduction. Hence, future work should investigate two directions: MALTASE-based failure reproduction with interaction traces containing noise as well as automated extraction of reproduction steps from monitored interaction traces, i.e. identification of the sub-trace which is necessary and sufficient to reproduce a failure.

Other researchers have proposed approaches to reproduce and fix field failures, especially capture/replay approaches and code monitoring approaches. Future work should compare such approaches with MALTASE-based failure reproduction. We argue that performance overhead introduced as well as the fraction of failures addressed by each approach are important aspects in this comparison.

Based on the positive evaluation results of MALTASE-based failure reproduction, we encourage developers to consider using such an approach in their applications. This consideration should include balancing the efforts to the benefits, designing an infrastructure similar to the MALTASE framework for monitoring and collection of user interactions preceding failures, and identification of circumstances in which users of their application accept such an approach.
Similarly, we encourage vendors of crash reporting tools to consider user interactions preceding failures, i.e. to monitor user interactions and include user interactions preceding a failure in failure reports together with memory dumps and stack traces.

The classifiers for MOSKitt skill clearly outperform random and ZeroR classifiers. Hence, we conclude that MALTASE-based skill detection can discriminate between novice and expert users of MOSKitt and that the MALTASE sensors provide enough information for this classification. Future work should investigate how this result translates to other applications.

The performance of classifiers for UML skill was unsatisfactory as most classifiers exhibit a similar performance as a ZeroR classifier (a classifier always predicting the most common skill level). We expected such a performance as the k means clustering

for UML skill was not able to separate participants according to their UML skill level. Hence, future work should investigate if it is possible to detect UML skill from user interactions and how classifiers for UML skill classifiers can be improved.

All decision trees required only a few classification features, indicating that only a few features suffice to detect user skill. This result confirms related work by Hurst et al. [84] who found that "differences between novice and expert users can be sensed with high accuracy using only a few key features". We conclude that skill classifiers introduce a small performance overhead as only a few features have to be monitored.

Classifiers learned from all classification features showed the same performance as classifiers learned from runtime features (features which can be extracted at runtime). This result indicates that non-runtime features do not provide information improving classifier performance. Hence, we conclude that classification features necessary for skill classifiers can be extracted at runtime and that runtime features suffice for skill classifiers.

We observed that task completion time is an indicator of high user skill for easy tasks but not for complex tasks. Similarly, we observed that hot key usage was an indicator for low user skill in the evaluation while it might indicate high user skill in other contexts. Hence, we conclude that the correlation of a classification feature with user skill depends on the context (task, application, domain) and that a careful analysis is necessary when transferring classifiers from one context to another one.

Detecting user skills raises ethical questions as this information might be misused. Hence, care should be taken to protect information about user skill such as anonymizing skill data or encrypting it during transfer. If the skill information is used only for adaptation of the user interface or the application, it should not be sent to a developer server.

We argue that information about user skills can be used in two ways: First, to automatically adapt the user interface or the application to the current user, e.g. by hiding complex features or providing additional help for novice users. And second, we hypothesize that developers can exploit information about the skill distribution of their user base in software evolution decisions. For example, if developers knew that the majority of their users are novices, they might decide to implement a certain functionality using a wizard style which provides user guidance. Future work should investigate such exploitations of user skill information in more detail.

Overall, the evaluation demonstrated that Maltase provides developers with helpful knowledge during software evolution, namely reproduction steps for failures and information about user skill. As Maltase is a framework, it can be extended by more sensors and more types of analysis to provide additional knowledge to developers. An example of a further analysis is the comparison of monitored user interactions to the flow of events of use cases to detect mismatches between both. Also, future work could investigate feature detection based on monitored user interactions, i.e. to identify which application features a user used during a session.
Different types of analyses performed on monitored, high-level user interactions show an advantage of this type of data: they can be reused, i.e. user interactions which have been monitored once can be analyzed multiple times to gain different kinds of information ("monitor once, analyze multiple times").

Based on evaluation results about the feasibility and impact of MALTASE, we encourage developers to consider using a usage analysis tool in their applications. This consideration should include an analysis of benefits of such an approach as well as balancing those benefits with the effort required for its installation and maintenance. Furthermore, developers should determine in which types of usage data they are interested in, how this data can be collected and analyzed, and how it impacts their software evolution tasks.

A prerequisite for using MALTASE is the acceptance by users of the target application. As the evaluation has shown the impact of MALTASE during software evolution, we encourage users to accept a usage analysis tool in the software applications they use. Care should be taken when designing a usage analysis tool to adhere to privacy concerns of users. To achieve this, we see the following measures: anonymizing monitored usage data, transferring usage data only with user approval, and presenting usage data to users to create transparency about collected data.
There is a dilemma regarding user acceptance for a usage analysis tool: Usually users do not have a direct benefit from a usage analysis tool. Hence, it might be difficult to convince them to use such a tool and future work could investigate incentives for motivating users to accept such a tool.

The evaluation was performed with one application. As a real world application was used, we argue that the evaluation results are relevant in practice. But future work could investigate whether the MALTASE framework can be employed with other applications and whether the same results hold. When integrating the MALTASE framework with an application, only the sensors and monitoring infrastructure component depend on the implementation technology and might have to be adapted or rewritten. The other parts of the the MALTASE framework are independent of the target application or its implementation technology.

6.6 Chapter Summary

This chapter described an empirical evaluation of the MALTASE framework. Framework components were implemented and integrated with MOSKitt, a real-world CASE tool. A simulation and a user survey were performed to investigate the performance overhead and user acceptance of MALTASE. The simulation simulation user sessions for a plain and an instrumented MOSKitt instance and compared performance metrics of both. It found that MALTASE monitoring introduces a performance overhead of 5 % execution time and 2-5 % memory consumption. During the user survey, six MOSKitt users worked with an instrumented MOSKitt instance for two weeks and reported their experiences. It found that the performance overhead introduced by MALTASE monitoring is not perceived by users or does not hinder them. Further, it found that participants accept anonymously interaction monitoring.

A controlled experiment was conducted to evaluate MALTASE-based failure reproduction and compare it to failure reproduction with textual bug reports. Participants had to reproduce failures from bug reports in the MOSKitt bug repository based on either monitored user interactions or textual bug reports. The experiment found that developers can reproduce failures based on monitored user interactions preceding the failure and that the MALTASE sensors provide enough information for

failure reproduction. Further, it found that MALTASE-based failure reproduction enables developers unfamiliar with an application to reproduce failures they could not reproduce with textual bug reports. Because of the limited number of experienced developers, we cannot draw conclusions for them but their positive feedback indicates an interest of developers in MALTASE-based failure reproduction.

An evaluation case study was performed to evaluate MALTASE-based skill detection. Participants had to work on UML modeling tasks in MOSKitt and their interactions were monitored by MALTASE sensors. This dataset was used to learn and evaluate decision tree classifiers for MOSKitt application skill and UML domain skill. The case study found that MALTASE-based skill detection can discriminate between novice and expert users of MOSKitt and that the MALTASE sensors provide enough information for this classification. The performance of classifiers for UML skill was similar to a classifier always predicting the most common skill level. Decision tree classifiers tested just a few classification features, indicating that skill classifiers have to introduce only a small performance overhead.

Overall, the evaluation showed that MALTASE provides developers with helpful knowledge during software evolution, namely reproduction steps for failures and information about user skills. Hence, MALTASE addresses the problem of missing usage information and narrows communication gaps between developers and users.

Chapter 7

Conclusion

This chapter summarizes the contributions of the dissertation (Section 7.1), discusses general issues and proposes future work (both in Section 7.2).

7.1 Contributions

This dissertation addressed the problem of missing usage knowledge among software developers. We hypothesized that such knowledge can be extracted from monitored, high-level user interactions. To study this hypothesis, we designed and implemented the MALTASE framework which monitors and analyzes high-level user interactions with the goal to exploit usage knowledge in software evolution. To evaluate the impact of MALTASE, we employed different research methods such as exploratory case study, simulation, user survey, controlled experiment, and evaluation case study. Our contribution is fourfold: First, we studied the dissertation problem in a problem case study and identified developer information needs regarding software usage. Second, we implemented the MALTASE framework to monitor and analyze high-level user interactions. We evaluated the performance overhead and user acceptance of MALTASE in a simulation and a user survey. Third, we demonstrated the viability of the MALTASE framework by implementing three applications of it. And fourth, we evaluated the impact of MALTASE in a controlled experiment and an evaluation case study. The remainder of this section presents these contributions in more detail.

Problem Case Study About Developer Information Needs

We conducted an exploratory problem case study about information needs of developers focussing on usage information. We analyzed observation protocols and interview minutes from sessions with 21 developers from six software companies during program comprehension tasks. We found that developers are interested in use cases and user behavior, user goals and user needs, failure reproduction steps, and application domain concepts. But such information is rarely available to them during software evolution. These findings complement related work about developer information needs. Also, we found that developers interact with the user interface of the target application to reproduce failures, to find relevant source code, to test implemented changes, to activate a debugger, and to familiarize themselves with unknown parts of the application. Overall, developers put themselves in the role of users by interacting with the user interface during program comprehension. We call this the "UI-based comprehension" and argue that this activity is part of a broader

comprehension strategy together with other activities like reading code or debugging. We summarize our results in a catalogue of 15 findings which can serve as starting point for further research.

Implementation and Evaluation of Maltase Framework

MALTASE is a framework for monitoring and analyzing high-level user interactions to gain relevant knowledge for developers. MALTASE contains components to monitor user interactions, store and transfer monitored data to a developer server, preprocess and mine monitored data, and present monitored data and analysis results to developers and integrate this information in the existing developer tool chain. MALTASE monitors user interactions on a high abstraction level such as user commands and artifact manipulations. MALTASE can be easily integrated into new target applications because it depend on toolkits and not on a particular application. MALTASE is extensible because it allows to add new sensors and analysis components.

We implemented the MALTASE framework and integrated it in the real-world UML editor MOSKitt to demonstrate its feasibility. Furthermore, we conducted a simulation and a user survey to investigate the performance overhead and user acceptance of MALTASE. The simulation found that MALTASE monitoring introduces a performance overhead of 5 % execution time and 2-5 % main memory consumption. In the user survey, we deployed MALTASE to six users for two weeks and collected their feedback in a survey. The user survey found that the performance overhead introduced by MALTASE is acceptable to users and that users accept that their interactions are monitored by MALTASE. MALTASE complements existing monitoring frameworks who either focus on monitoring code execution or exploit monitored user interactions for usability evaluation or usage statistics.

Applications of Maltase Framework in Software Evolution

We described three applications of the MALTASE framework: presenting monitored user interactions preceding failures to developers as reproduction steps, classification of user skills based on monitored user interactions, and comparison of monitored user interactions to use case steps. These applications illustrate how MALTASE can be employed to provide developers with usage knowledge. In addition, different framework applications demonstrate the generalizability of MALTASE: the same data - monitored user interactions - can be exploited for different purposes.

Evaluation of Maltase Applications

We investigated the impact of MALTASE by evaluating two framework applications in detail. To evaluate MALTASE-based failure reproduction, we conducted a controlled experiment comparing MALTASE-based failure reproduction to failure reproduction using textual bug reports submitted by users. The controlled experiment found that developers can elicit reproduction steps from monitored and visualized high-level user interactions. MALTASE-based failure reproduction enables inexperienced developers to reproduce failures they cannot reproduce with bug reports submitted by users. Because of a limited number of participants, we cannot draw conclusions

for experienced developers. But the positive feedback of participating developers indicates a strong interest of developers.

To evaluate MALTASE-based skill detection, we conducted an evaluation case study learning skill classifiers from monitored interactions of participants with differing skill levels. The case study found that MALTASE-based skill detection can discriminate between novice and expert users of MOSKitt. The performance of classifiers for UML Class Diagram skill was unsatisfactory and requires future research.

As MALTASE enables inexperienced developers to reproduce previously unreproducible failures and is able to discriminate between expert and novice MOSKitt users, it provides developers with helpful knowledge during software evolution. We conclude that our main hypothesis is true. Furthermore, we conclude that MALTASE addresses the problem of missing usage information and narrows the communication gap between developers and users.

Parts of this dissertation have been published in the following papers: [76], [137], [149], [150], [151], [152], [153], [154].

7.2 General Discussion and Future Work

Extension and Improvement of Maltase Framework

As with each approach, MALTASE has advantages and disadvantages. We see the following advantages of MALTASE: it requires no effort from users, it requires only little effort from developers for analyzing monitored interactions and analysis results, it analyzes real user behavior, and it is scalable to a large number of users because if its automation. On the other hand, we see the following disadvantages of MALTASE: it captures no information about user goals, motivation behind monitored behavior, or visionary feedback like feature requests. Hence, MALTASE is suitable to gain insights about the current state of the target application, but not about visionary future improvements as expressed in feature requests. Furthermore, developers might want to investigate the motivation behind observed user behavior which is usually not possible by analyzing usage data alone. To address these limitations, future work could investigate how MALTASE can be combined with other user involvement methods such user observations, user interviews, app store feedback, bug reports, or integrated feedback mechanisms.

The integration of MALTASE into the tool chain of developers could be improved. Currently, the Report Generator component generates bug reports with monitored user interactions and injects them into a bug repository. A mechanism to aggregate bug reports of the same origin could avoid flooding the bug repository. Furthermore, the visualization component could be extended to show bug reports with preceding user interactions, user skill statistics, and detected differences between monitored interactions and use case steps in an integrated fashion. We envision a dashboard-like presentation of monitored data and analysis results as a single access point for usage knowledge. Such a dashboard could be integrated into Integrated Development Environments or be deployed as a separate Web application.

We hypothesize that informing users about the interaction monitoring and user control over the sensors are important measures for the user acceptance of Maltase. Currently, the monitoring is performed in the background and hidden from the user. The Maltase framework could be extended to enable users to inspect monitored data and control the sensors. Future work in this direction could investigate how monitored data can be presented to users in an understandable way.

Maltase-based failure reproduction could be improved to deal with noisy interactions automatically, i.e. to automatically identify the subsequence of monitored interaction traces needed to reproduce failures. One could use delta debugging [202] or sequential pattern mining [118] for this purpose. Also, future research could investigate how the performance of UML skill classifiers in Maltase-based skill detection can be improved. Maltase-based use case testing could be improved by relieving the strict constraints on the order of use case steps. Future work could investigate approaches which allow the comparison of monitored user interactions and use case steps without placing such strict restrictions. Furthermore, future work could investigate approaches to automatically split interaction traces into sessions corresponding to a use case instance and to detect the current use case.

Additionally to improving existing framework applications, future work could investigate new applications. Maltase can possibly identify usability problems, e.g. by monitoring backtracking user interactions indicating usability problems as proposed by Akers et al. [5]. Similarly, Maltase can possibly detect which software feature a user is using and to calculate feature usage statistics.

This dissertation focused on analyzing individual user interaction traces. Another research direction is to investigate the analysis of sets of interaction traces. Analyzing sets of interaction traces can possibly reveal common and uncommon user behavior within a user group or detect changes in the behavior of a particular user.

Similarly, this dissertation focused on monitoring high-level interactions of user with a WIMP GUI of a desktop application. We hypothesize that Maltase can be applied to other types of interactive applications like e.g. mobile apps. Future work could investigate this hypothesis and study which changes of Maltase are necessary for such a situation.

Further Evaluation of Maltase Framework

The following parts of the Maltase framework have not yet been evaluated: the Undo History Extractor component and Maltase-based use case testing. Evaluation of the Undo History Extractor could investigate whether it is possible to elicit reproduction steps, user skills, or deviations of user behavior from use case steps from the interaction history of the undo feature. Evaluation of Maltase-based use case testing should investigate whether detected differences allow developers to identify software improvements and use case updates.

This dissertation investigated the use of Maltase during software evolution. But we see two additional usage scenarios of Maltase: software testing and prototype-based software development.

Software testing with Maltase requires to integrate Maltase with a version of the target application to be tested and to monitor the behavior of testers. Testers can be dedicated in-house testing personnel or potential users which participate in a beta

test. All three Maltase applications can be employed during testing: Maltase-based failure reproduction can be used to document failures and their reproduction steps which are detected by testers, Maltase-based skill detection can be used to investigate if the testers represent the target user population with regard to user skill, and Maltase-based use case testing can be used to get feedback where testers deviate from expected behavior and to identify potential software improvements. Prototype-based software development with Maltase requires the integration of Maltase with a prototype of the target application and to conduct user studies during development. Technologies such as continuous integration and continuous delivery ensure that a usable version of the application is available at any time and enable this usage scenario. Prototype-based software development with Maltase allows developers to get feedback about software usage early and to consider this feedback in future development activities. The Maltase applications can be employed in the following ways: Maltase-based failure reproduction can be used to document failures and their reproduction steps and Maltase-based use case testing to detect where users deviate from expected behavior and identify software improvements. Future work could investigate these usage scenarios.

This dissertation focused on the acquisition of usage knowledge from monitored user interactions. Future work could study how the exploitation of usage knowledge can be integrated in software development and evolution processes, e.g. during activities like sprint planning. We refer to related work by van der Schuur et al. [191] and Krusche et al. [103] who study such aspects.

Further Research Directions

This dissertation investigated Maltase and usage monitoring in general mainly from a developer perspective. To comprehensively evaluate Maltase and make it usable in practice, it is necessary to study the user perspective as well. We see two main aspects which could be investigated by future work: user privacy and indirect user benefit. As Maltase monitors user interactions, privacy issues arise as sensitive information might be captured. An important aspect for future investigation is to study under which circumstances users accept Maltase, i.e. which information about their software usage they are willing to share with which organizations and for which purposes. A recent survey by Sheth et al. [166] as well as the user survey in this dissertation indicate that users perceive interaction data as less sensitive than personal data or document content. But future research could investigate privacy issues during user monitoring in more detail. While Maltase provides a direct benefit for developers because of the acquisition of usage knowledge, its benefit for software users is indirect: In exchange for using a target application instrumented with Maltase, they are promised an improved target application with less bugs, less usability problems, and functionality tailored to their needs. Future work could investigate whether this promise motivates users to accept Maltase, how the benefits of Maltase can be explained to users, and how users can be motivated to use a target application instrumented with Maltase.

The problem case study identified "UI-based comprehension", a comprehension activity during which developers interact with the user interface during program comprehension. As this comprehension activity has not been studied in detail,

future work could investigate it. In particular, it could investigate the motivation of developers behind this behavior, the context in which developers employ this behavior, the combination with other comprehension activities, and possibilities for tool support. Furthermore, the problem case study identified developer information needs similarly to several other empirical studies [14, 15, 22, 26, 54, 96, 117, 168]. Future work could develop a theory of developer information needs based on these studies. We argue that it would be interesting to contextualize information needs, i.e. to determine the development context such as the task or the type of application in which a particular information need arises.

Throughout this dissertation, we used the term "maltase" as abbreviation for "Monitoring, Analysis, and ExpLoiTation of User InterActions in Software Evolution". But originally, maltase is the name of an enzyme that breaks disaccharide maltose (malt sugar) and enables humans to digest carbohydrates. In this spirit, we hope that MALTASE enables developers to "digest" user interaction traces and usage data to extract helpful usage knowledge and to bridge the communication gap between developers and users.

Bibliography

[1] IEEE Standard for Software Maintenance. *IEEE Std 1219-1998*, 1998.

[2] U. Abelein and B. Paech. A proposal for enhancing user-developer communication in large IT projects. In *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 1–3. IEEE, 2012.

[3] U. Abelein and B. Paech. State of Practice of User-Developer Communication in Large-Scale IT Projects: Results of an Expert Interview Series. In C. Salinesi and I. van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, volume 8396 of *Lecture Notes in Computer Science*, pages 95–111. Springer, 2014.

[4] U. Abelein and B. Paech. Understanding the Influence of User Participation and Involvement on System Success - a Systematic Mapping Study. *Empirical Software Engineering*, 20(1):28–81, 2015.

[5] D. Akers, R. Jeffries, M. Simpson, and T. Winograd. Backtracking events as indicators of usability problems in creation-oriented applications. *ACM Transactions on Computer-Human Interaction*, 19(2):1–40, 2012.

[6] J. Alexander, A. Cockburn, and R. Lobb. AppMonitor: A tool for recording user actions in unmodified Windows applications. *Behavior Research Methods*, 40(2):413–421, 2008.

[7] R. Ali, C. Solis, M. Salehie, I. Omoronyia, B. Nuseibeh, and W. Maalej. Social sensing: when users become monitors. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 476–479. ACM, 2011.

[8] G. Antonio, D. Lucca, A. R. Fasolino, U. D. Carlini, N. Federico, and V. Claudio. Recovering use case models from object-oriented code: a thread-based approach. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 108–117. IEEE, 2000.

[9] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP 2008 - Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565. Springer, 2008.

[10] M. Bano and D. Zowghi. A systematic review on the relationship between user involvement and system success. *Information and Software Technology*, 58(0):148–169, 2015.

[11] G. Baster, P. Konana, and J. Scott. Business Components: A Case Study of Bankers Trust Australia Limited. *Communications of the ACM*, 44(5):92–98, 2001.

[12] S. Bateman, C. Gutwin, N. Osgood, and G. McCalla. Interactive usability instrumentation. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 45–54. ACM, 2009.

[13] R. Beale, J. Finlay, J. Austin, and M. Harrison. User modelling in classification: a neural-based approach. In J. Taylor and C. Mannion, editors, *New Developments in Neural Computing*, pages 103–110. Adam Hilger LTD, 1989.

[14] A. Begel, K. Y. Phang, and T. Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 125–134. ACM, 2010.

[15] A. Begel and T. Zimmermann. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23. ACM, 2014.

[16] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 362–371. ACM, 2013.

[17] M. Bezold and W. Minker. User Modeling in Interactive Systems. In *Adaptive Multimodal Interactive Systems*, pages 41–65. Springer, 2011.

[18] E. Bjarnason, K. Wnuk, and B. Regnell. Requirements are slipping through the gaps - A case study on causes & effects of communication gaps in large-scale software development. In *Proceedings of the 19th IEEE International Requirements Engineering Conference*, pages 37–46. IEEE, 2011.

[19] B. Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.

[20] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu. Using formal verification to evaluate human-automation interaction: A review. *IEEE Transactions on Systems, Man, and Cybernetics*, 43(3):488–503, 2013.

[21] M. L. Bolton, N. Jim, M. M. V. Paassen, and M. Trujillo. From Task Models for the Formal Verification of Human-Automation Interaction. *IEEE Transactions on Human-Machine Systems*, 44(5):561–575, 2014.

[22] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative work*, pages 301–310. ACM, 2010.

[23] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* Pearson Education, 3rd edition, 2010.

[24] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, pages 473–484. ACM, 2013.

[25] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley and Sons, 1996.

[26] R. P. L. Buse and T. Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*, pages 987–996. IEEE, 2012.

[27] Y. Cao, H. Zhang, and S. Ding. SymCrash: Selective Recording for Reproducing Crashes. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 791–802. ACM, 2014.

[28] N. Chapin, J. E. Hale, K. Khan, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[29] L. C. L. Chen and Q. L. Q. Li. Automated test case generation from use case: A model based approach. In *Proceedings of the 3rd IEEE International Conference on Computer Science and Information Technology*, pages 372–377. IEEE, 2010.

[30] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proceedings of the 29th International Conference on Software Engineering*, pages 261–270. ACM, 2007.

[31] J. Clause and A. Orso. Penumbra: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA'09, pages 249–259. ACM, 2009.

[32] M. Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[33] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[34] C. L. Corritore and S. Wiedenbeck. Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 139–148. IEEE, 2000.

[35] J. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE Publications, 2009.

[36] A. L. L. de Figueiredo, W. L. Andrade, and P. D. L. Machado. Generating interaction test cases for mobile phone systems from use case specifications. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–10, 2006.

[37] A. K. Dey, G. D. Abowd, and D. Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context Aware Applications. *Human-Computer Interaction*, 16(2):97–166, 2001.

[38] M. M. Diep. Analysis of a Deployed Software. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 595–598. ACM, 2007.

[39] E. W. Dijkstra. Software Engineering As It Should Be (Panel Remark). In *Proceedings of the 4th International Conference on Software Engineering*. IEEE, 1979.

[40] A. Dix, J. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, 3rd edition, 2003.

[41] M. Dostál and Z. Eichler. A Hybrid Approach to User Activity Instrumentation in Software Applications. In *HCI International 2011 - Posters' Extended Abstracts*, volume 173 of *Communications in Computer and Information Science*, pages 566–570. Springer, 2011.

[42] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting Empirical Methods for Software Engineering Research. In F. Shull, J. Singer, and D. I. Sjø berg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.

[43] M. El-Ramly and E. Stroulia. Mining software usage data. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, 2004.

[44] M. El-Ramly, E. Stroulia, and P. Sorenson. From run-time behavior to usage scenarios: An interaction-pattern mining approach. In *Proceedings of the eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 315–324. ACM, 2002.

[45] M. El-Ramly, E. Stroulia, and P. Sorenson. Mining system-user interaction traces for use case models. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 21–29. IEEE, 2002.

[46] M. El-Ramly, E. Stroulia, and P. Sorenson. Recovering software requirements from system-user interaction traces. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 447–454. ACM, 2002.

[47] K. Ericsson and H. Simon. *Protocol analysis: Verbal reports as data (rev. ed.)*. MIT Press, 1993.

[48] M. Feather, S. Fickas, a. V. Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, 1998.

[49] K. Fenstermacher and M. Ginsburg. A lightweight framework for cross-application user monitoring. *IEEE Computer*, 35(3):51–59, 2002.

[50] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical Studies of Open Source Evolution. In *Software Evolution*, pages 263–288. Springer, 2008.

[51] S. Feuerstack, M. Blumendorf, M. Kern, M. Kruppa, M. Quade, M. Runge, and S. Albayrak. Automated usability evaluation during model-based interactive system development. In *Engineering Interactive Systems*, volume 5247 of *Lecture Notes in Computer Science*, pages 134–141. Springer, 2008.

[52] S. Fickas and M. S. Feather. Requirements Monitoring in Dynamic Environments. In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, pages 140–147. IEEE, 1995.

[53] G. Fischer. User Modeling in Human Computer Interaction. *User Modeling and User-Adapted Interaction*, 11(1-2):65–86, 2001.

[54] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184. ACM, 2010.

[55] M. Funk, P. Hoyer, and S. Link. Model-driven instrumentation of graphical user interfaces. In *Proceedings of the 2nd International Conference on Advances in Computer-Human Interaction*, pages 19–25. IEEE, 2009.

[56] A. R. Gagné, M. Seif El-Nasr, and C. D. Shaw. Analysis of Telemetry Data from a Real-time Strategy Game: A Case Study. *Computers in Entertainment*, 10(3):1–25, 2012.

[57] L. Galvis Carreno and K. Winbladh. Analysis of User Comments: An Approach for Software Requirements Evolution. In *Proceedings of the 35th International Conference on Software Engineering*, pages 582–591. ACM, 2013.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994.

[59] J. J. Garrett. *The Elements of User Experience: User-Centered Design for the Web and Beyond*. Pearson Education, 2010.

[60] A. Ghazarian and A. Ghazarian. Pauses in man-machine interactions: a clue to users' skill levels and their user interface requirements. *International Journal of Cognitive Performance Support*, 1(1):82–102, 2013.

[61] A. Ghazarian and S. Noorhosseini. Automatic detection of users' skill levels using high-frequency user interface events. *User Modeling and User-Adapted Interaction*, 20(2):109–146, 2010.

[62] A. Girgensohn, D. F. Redmiles, and F. M. Shipman. Agent-Based Support for Communication between Developers and Users in Software Design. In *Proceedings of the 9th Conference on Knowledge-Based Software Engineering*, pages 22–29. IEEE, 1994.

[63] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pages 103–116. ACM, 2009.

[64] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35th International Conference on Software Engineering*, pages 72–81. IEEE, 2013.

[65] K. Grindley. Managing IT at board level: The hidden agenda exposed. *Pitmans, London*, 1991.

[66] J. Grudin. Interactive Svstems: Bridging the Gaps between Developers and Users. *Computer*, 24(4):59–69, 1991.

[67] N. Gurbanova. Presenting User and Context Information to Developers during Bug Fixing (Master Thesis), 2012.

[68] E. Guzman and W. Maalej. How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference*, pages 153–162. IEEE, 2014.

[69] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[70] M. A. Harris and H. R. Weistroffer. A new look at the relationship between user involvement in systems development and system success. *Communications of the Association for Information Systems*, 24(1), 2009.

[71] G. S. Hartman and L. Bass. Logging Events Crossing Architectural Boundaries. In M. Costabile and F. Paternò, editors, *Human-Computer Interaction - INTERACT 2005*, volume 3585 of *Lecture Notes in Computer Science*, pages 823–834. Springer, 2005.

[72] B. Hasling, H. Goetz, and K. Beetz. Model Based Testing of System Requirements using UML Use Case Models. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pages 367–376. IEEE, 2008.

[73] J. Heiskari and L. Lehtola. Investigating the State of User Involvement in Practice. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference*, pages 433–440. IEEE, 2009.

[74] S. Herbold. *Usage-based Testing for Event-driven Software*. PhD thesis, 2012.

[75] S. Herbold, J. Grabowski, S. Waack, and U. Bünting. Improved bug reporting and reproduction through non-intrusive GUI usage monitoring and automated replaying. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops*, pages 232–241. IEEE, 2011.

[76] T.-M. Hesse, S. Gärtner, T. Roehm, B. Paech, K. Schneider, and B. Bruegge. Semiautomatic Security Requirements Engineering and Evolution using Decision Documentation, Heuristics, and User Monitoring. In *Proceedings of the 1st International Workshop on Evolving Security and Privacy Requirements Engineering*, pages 1–6. IEEE, 2014.

[77] D. Hilbert and D. Redmiles. Agents for collecting application usage data over the Internet. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 149–156. ACM, 1998.

[78] D. M. Hilbert and D. F. Redmiles. An Approach to Large-scale Collection of Application Usage Data Over the Internet. In *Proceedings of the 20th International Conference on Software Engineering*, pages 136–145. IEEE, 1998.

[79] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, 2000.

[80] D. M. Hilbert and D. F. Redmiles. Large-Scale Collection of Usage Data to Inform Design. In *Proceedings of the Eighth IFIP Conference on Human-Computer Interaction*, pages 569–576, 2001.

[81] D. E. Hoiem and K. D. Sullivan. Designing and using integrated data collection and analysis tools: challenges and considerations. *Behaviour & Information Technology*, 13(1-2):160–170, 1994.

[82] S. Hornik, G. Klein, and J. Jiang. Communication skills of IS providers: an expectation gap analysis from three stakeholder perspectives. *IEEE Transactions on Professional Communication*, 4(1):17–34, 2003.

[83] K. Hullett, N. Nagappan, E. Schuh, and J. Hopson. Empirical analysis of user data in game software development. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 89–98. ACM, 2012.

[84] A. Hurst, S. E. Hudson, and J. Mankoff. Dynamic detection of novice vs. skilled use without a task model. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 271–280. ACM, 2007.

[85] M. Y. Ivory and M. a. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4):470–516, 2001.

[86] W. Jiang, H. Ruan, and L. Zhang. Analysis of Economic Impact of Online Reviews: An Approach for Market-Driven Requirements Evolution. In D. Zowghi and Z. Jin, editors, *Requirements Engineering*, volume 432 of *Communications in Computer and Information Science*, pages 45–59. Springer, 2014.

[87] W. Jiang, H. Ruan, L. Zhang, P. Lew, and J. Jiang. For User-Driven Software Evolution: Requirements Elicitation Derived from Mining Online Reviews. In V. Tseng, T. Ho, Z.-H. Zhou, A. Chen, and H.-Y. Kao, editors, *Advances in Knowledge Discovery and Data Mining*, volume 8444 of *Lecture Notes in Computer Science*, pages 584–595. Springer, 2014.

[88] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 474–484. IEEE, 2012.

[89] W. Jin and A. Orso. F3: fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 213–223. ACM, 2013.

[90] P. Johnson. Project Hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis. Technical report, University of Hawai, 2001.

[91] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 234–243. IEEE, 2007.

[92] J. Kay and R. C. Thomas. Studying long-term system use. *Communications of the ACM*, 38(7):61–69, 1995.

[93] M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11. ACM, 2006.

[94] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella. Reproducing Field Failures for Programs with Complex Grammar-Based Input. In *Proceedings of the Seventh IEEE International Conference on Software Testing, Verification and Validation*, pages 163–172. IEEE, 2014.

[95] J. H. Kim, D. V. Gunn, E. Schuh, B. C. Phillips, R. J. Pagulayan, and D. Wixon. Tracking real-time user experience (TRUE): A comprehensive instrumentation solution for complex systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 443–451. ACM, 2008.

[96] A. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International conference on Software Engineering*, pages 344–353. IEEE, 2007.

[97] A. Ko, B. Myers, M. Coblenz, and H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[98] A. J. Ko, M. J. Lee, V. Ferrari, S. Ip, and C. Tran. A Case Study of Post-Deployment User Feedback Triage. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 1–8. ACM, 2011.

[99] A. J. Ko and B. Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 175–184. IEEE, 2003.

[100] J. Koenemann and S. P. S. Robertson. Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 125–130. ACM, 1991.

[101] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, and E.-J. Malm. Managing context information in mobile devices. *IEEE Pervasive Computing*, 2(3):42–51, 2003.

[102] B. Kristjánsson and H. van der Schuur. A Survey of Tools for Software Operation Knowledge Acquisition. Technical report, Utrecht University, 2009.

[103] S. Krusche, L. Alperowitz, B. Bruegge, and M. O. Wagner. Rugby: An Agile Process Model Based on Continuous Delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 42–50. ACM, 2014.

[104] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models. In *Proceedings of the 28th international Conference on Software Engineering*, pages 492–501. ACM, 2006.

[105] E. I. Laukkanen and M. V. Mantyla. Survey Reproduction of Defect Reporting in Industrial Software Development. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 197–206. IEEE, 2011.

[106] A. Lecerof and F. Paternò. Automatic Support for Usability Evaluation. *IEEE Transactions on Software Engineering*, 24(10):863–888, 1998.

[107] Q. Li, S. Hu, P. Chen, L. Wu, and W. Chen. Discovering and mining use case model in reverse engineering. In *Proceedings of the 4th International Conference on Fuzzy Systems and Knowledge Discovery*, pages 431–436. IEEE, 2007.

[108] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Notices*, 38(5):141, 2003.

[109] F. Linton, D. Joy, H. P. Schaefer, and A. Charron. OWL: A recommender system for organization-wide learning. *Educational Technology and Society*, 3(1):62–76, 2000.

[110] D. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental Models and Software Maintenacne. *Journal of Systems and Software*, 7(4):341–355, 1987.

[111] J. Liu, K. Wong, and K. Hui. Discovering User Behavior Patterns in Personalized Interface Agents. In K. Leung, L.-W. Chan, and H. Meng, editors, *Intelligent Data Engineering and Automated Learning - IDEAL 2000. Data Mining, Financial Engineering, and Intelligent Agents*, volume 1983 of *Lecture Notes in Computer Science*, pages 398–403. Springer, 2000.

[112] W. Maalej. *Intention-Based Integration of Software Engineering Tools*. Phd thesis, 2010.

[113] W. Maalej, T. Fritz, and R. Robbes. Collecting and Processing Interaction Data for Recommendation Systems. In M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 173–197. Springer, 2014.

[114] W. Maalej and H.-J. Happel. A Lightweight Approach for Knowledge Sharing in Distributed Software Teams. In T. Yamaguchi, editor, *Practical Aspects of Knowledge Management*, volume 5345 of *Lecture Notes in Computer Science*, pages 14–25. Springer, 2008.

[115] W. Maalej, H.-J. Happel, and R. Asarnusch. When users become collaborators: Towards continuous and context-aware user input. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 981–990. ACM, 2009.

[116] W. Maalej and D. Pagano. On the socialness of software. In *Proceedings of the Ninth IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 864–871. IEEE, 2011.

[117] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the Comprehension of Program Comprehension. *ACM Transactions on Software Engineering and Methodology*, 23(4):31:1–31:37, 2014.

[118] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):1–41, 2010.

[119] J. Mann. IT Education's Failure to Deliver Successful Information Systems: Now is the Time to Address the IT-User Gap. *Journal of Information Technology Education*, 1(1):253–267, 2002.

[120] J. Matejka, W. Li, T. Grossman, and G. Fitzmaurice. CommunityCommands: Command Recommendations for Software Applications. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, pages 193–202. ACM, 2009.

[121] T. Menzies and T. Zimmermann. Software Analytics: So What? *IEEE Software*, 30(4):31–37, 2013.

[122] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[123] E. Murphy-Hill, N. Carolina, R. Jiresal, and G. C. Murphy. Improving Software Developers' Fluency by Recommending Development Environment Commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 42:1–42:11. ACM, 2012.

[124] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[125] H. Nagasaki and M. Azuma. A methodology for assessing user's skill grade to implement adaptive user interface systems. In *Proceedings. of the First IEEE International Conference on Cognitive Informatics*, pages 280–287. IEEE, 2002.

[126] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel. Automatic test generation: a use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.

[127] J. Nielsen. *Usability Enginering*. Morgan Kaufmann Publishers Inc., 1993.

[128] J. Nielsen. Alertbox: First Rule of Usability? Don't Listen to Users., 2001.

[129] D. Norman. *The Design of Everyday Things*. Basic Books, 2013.

[130] J. Oh, D. Kim, U. Lee, J.-g. Lee, and J. Song. Facilitating Developer-User Interactions with Mobile App Review Digests. In *Extended Abstracts on Human Factors in Computing Systems*, pages 1809–1814. ACM, 2013.

[131] A. Orso. Monitoring, Analysis, and Testing of Deployed Software. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 263–267. ACM, 2010.

[132] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proceedings of the 3rd International Workshop on Dynamic Analysis*, pages 1–7. ACM, 2005.

[133] S. Pachidi, M. Spruit, and I. van de Weerd. Understanding users' behavior with software operation data mining. *Computers in Human Behavior*, 30(0):583–594, 2014.

[134] D. Pagano. Towards Systematic Analysis of Continuous User Input. In *Proceedings of the 4th International Workshop on Social Software Engineering*, pages 7–11. ACM, 2011.

[135] D. Pagano. *PORTNEUF - A Framework for Continuous User Involvement*. PhD thesis, 2013.

[136] D. Pagano and B. Bruegge. User Involvement in Software Evolution Practice: A Case Study. In *Proceedings of the 35th International Conference on Software Engineering*, pages 953–962. IEEE, 2013.

[137] D. Pagano, M. Juan, A. Bagnato, T. Roehm, B. Bruegge, and W. Maalej. FastFix: Monitoring control for remote software maintenance. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1437–1438. IEEE, 2012.

[138] D. Pagano and W. Maalej. User Feedback in the AppStore: An Empirical Study. In *Proceedings of the 21st IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, 2013.

[139] F. Paternò, A. Piruzza, and C. Santoro. Remote web usability evaluation exploiting multimodal information on user behavior. In *Computer-Aided Design of User Interfaces V*, pages 287–298. Springer, 2007.

[140] K. Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer, 2010.

[141] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[142] N. A. Qureshi, N. Seyff, and A. Perini. Satisfying User Needs at the Right Time and in the Right Place: A Research Preview. In D. Berry and X. Franch, editors, *Requirements Engineering: Foundation for Software Quality*, volume 6606 of *Lecture Notes in Computer Science*, pages 94–99. Springer, 2011.

[143] D. F. Redmiles. Supporting the end users' views. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 34–42. ACM, 2002.

[144] B. Regnell, P. Runeson, and C. Wohlin. Towards integration of use case modelling and usage-based testing. *Journal of Systems and Software*, 50(2):117–130, 2000.

[145] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[146] W. N. Robinson. Seeking Quality through User-Goal Monitoring. *IEEE Software*, 26(5):58–65, 2009.

[147] W. N. Robinson. A roadmap for Comprehensive Requirements Monitoring. *IEEE Software*, 43(5):64–72, 2010.

[148] W. N. Robinson. Understanding Software System Evolution through Requirements Monitoring. *Requirements Management-Novel Perspectives and Challenges*, 2012.

[149] T. Roehm. Two User Perspectives in Program Comprehension: End Users and Developer Users. In *Proceedings of the 23th IEEE International Conference on Program Comprehension (Submitted)*, 2015.

[150] T. Roehm and B. Bruegge. Reproducing Software Failures By Exploiting the Action History of Undo Features. In *Proceedings of the 36th International Conference on Software Engineering*, pages 496–499. ACM, 2014.

[151] T. Roehm, B. Bruegge, T.-M. Hesse, and B. Paech. Towards Identification of Software Improvements and Specification Updates By Comparing Monitored and Specified End User Behavior. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 464–467. IEEE, 2013.

[152] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej. Monitoring User Interactions for Supporting Failure Reproduction. In *Proceedings of the 21th IEEE International Conference on Program Comprehension*, pages 73–82. IEEE, 2013.

[153] T. Roehm and W. Maalej. Automatically Detecting Developer Activities and Problems in Software Development Work. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1261–1264. IEEE, 2012.

[154] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering*, pages 255–265. IEEE, 2012.

[155] Y. Rogers, H. Sharp, and J. Preece. *Interaction Design*. John Wiley & Sons, 3rd edition, 2011.

[156] R. L. Rosnow and R. Rosenthal. *Beginning behavioral research: A conceptual primer*. Pearson, 6th edition, 1996.

[157] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2008.

[158] M. a. Sao Pedro, R. S. J. De Baker, J. D. Gobert, O. Montalvo, and A. Nakama. Leveraging machine-learned detectors of systematic inquiry behavior to estimate and predict transfer of inquiry skill. *User Modelling and User-Adapted Interaction*, 23(1):1–39, 2013.

[159] B. Schinzel. Weltbilder und Bilder der Informatik. *Informatik-Spektrum*, 36(3):260–266, 2013.

[160] F. Schlesinger and S. Jekutsch. ElectroCodeoGram: An Environment for Studying Programming. In *Workshop on Ethnographies of Code*, pages 30–31, 2006.

[161] K. Schneider, S. Meyer, M. Peters, F. Schliephacke, J. Mörschbach, L. Aguirre, and S. M. De. Feedback in Context: Supporting the Evolution of IT-Ecosystems. In M. Ali Babar, M. Vierimaa, and M. Oivo, editors, *Product-Focused Software Process Improvement*, volume 6156 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2010.

[162] N. Seyff, F. Graf, and N. Maiden. Using Mobile RE Tools to Give End-Users Their Own Voice. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*, pages 37–46. IEEE, 2010.

[163] N. Seyff, G. Ollmann, and M. Bortenschlager. AppEcho: A User-Driven, In Situ Feedback Approach for Mobile Platforms and Applications. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, pages 99–108. ACM, 2014.

[164] T. M. Shaft and I. Vessey. The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process. *Journal of Management Information Systems*, 15(1):51–78, 1998.

[165] S. Shekh and S. Tyerman. An Aspect-Oriented Framework for Event Capture and Usability Evaluation. In L. Maciaszek, C. González-Pérez, and S. Jablonski, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 69 of *Communications in Computer and Information Science*, pages 107–119. Springer, 2010.

[166] S. Sheth, G. Kaiser, and W. Maalej. Us and Them - A Study of Privacy Requirements Across North America, Asia, and Europe. In *Proceedings of the 36th International Conference on Software Engineering*, pages 859–870. ACM, 2014.

[167] J. Sillito, K. De Volder, B. Fisher, and G. Murphy. Managing software change tasks: an exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 23–32. IEEE, 2005.

[168] J. Sillito, G. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[169] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An Examination of Software Engineering Work Practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 174–188. IBM Press, 1997.

[170] M. Smit, E. Stroulia, and K. Wong. Use case redocumentation from GUI event traces. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 263–268. IEEE, 2008.

[171] D. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the Start user interface. *Byte*, 7(4):242–282, 1982.

[172] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/ replay tool for observation-based testing. *ACM SIGSOFT Software Engineering Notes*, 25(5):158–167, 2000.

[173] E. Stroulia, M. El-Ramly, P. Iglinski, and P. Sorenson. User interface reverse engineering in support of interface migration to the web. *Automated Software Engineering*, 10(3):271–301, 2003.

[174] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuk. Reverse engineering legacy interfaces: an interaction-driven approach. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 292–302. IEEE, 1999.

[175] E. Stroulia, M. El-Ramly, and P. Sorenson. From legacy to Web through interaction modeling. In *Proceedings of the International Conference on Software Maintenance*, pages 320–329. IEEE, 2002.

[176] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[177] Y. Tao. Capturing user interface events with aspects. In *Human-Computer Interaction. HCI Applications and Services*, volume 4553 of *Lecture Notes in Computer Science*, pages 1170–1179. Springer, 2007.

[178] Y. Tao. Toward Computer-Aided Usability Evaluation for Evolving Interactive Software. In *Proceedings of the Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2007.

[179] Y. Tao. Grammatical Analysis of User Interface Events for Task Identification. In A. Marcus, editor, *Design, User Experience, and Usability. Theories, Methods, and Tools for Designing the User Experience*, volume 8517 of *Lecture Notes in Computer Science*, pages 197–205. Springer, 2014.

[180] A. M. Tarta and G. Moldovan. Automatic Usability Evaluation Using AOP. In *Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 84–89. IEEE, 2006.

[181] A. Taylor-Cummings and D. Feeny. The Development and Implementation of Systems: Bridging the User-IS Gap. In L. Willcocks, D. Feeny, and G. Islei, editors, *Managing IT as a strategic resource*, pages 171–198. McGraw-Hill, 1997.

[182] M. Terry, M. Kay, B. V. Vugt, B. Slack, T. Park, and B. Van Vugt. ingimp: Introducing instrumentation to an end-user open source application. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 607–616. ACM, 2008.

[183] D. Tesch, M. G. Sobol, G. Klein, and J. J. Jiang. User and developer common knowledge: Effect on the success of information system development projects. *International Journal of Project Management*, 27(7):657–664, 2009.

[184] S. Theiner. User Skill Classification Based on Interaction Trace Analysis (Master Thesis), 2013.

[185] J. Trumbly, A. Kirk, and M. Merle. Performance effect of matching computer interface characteristics and user skill level. *International Journal of Man-Machine Studies*, 338(4):713–724, 1993.

[186] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of 21th ACM SIGOPS Symposium on Operating Systems Principles*, pages 131–144. ACM, 2007.

[187] D. Tuffley. Exploring the IT-User Gap: Towards developing communication strategies. In *Qualitative Research in IT & IT in Qualitative Research*, 2005.

[188] D. Tuffley. *Software Requirements: Closing The User-Developer Gap - Technical Writer as Facilitator between User and Developer During the Software Requirements Analysis Phase*. VDM Verlag, Saarbrücken, 2008.

[189] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[190] H. Van Der Schuur, S. Jansen, and S. Brinkkemper. A reference framework for utilization of software operation knowledge. In *Proceedings of the36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 245–254. IEEE, 2010.

[191] H. Van Der Schuur, S. Jansen, and S. Brinkkemper. If the SOK fits, wear it: Pragmatic process improvement through software operation knowledge. In D. Caivano, M. Oivo, M. Baldassarre, and G. Visaggio, editors, *Product-Focused Software Process Improvement*, volume 6759 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2011.

[192] K. P. Vaubel and C. F. Gettys. Inferring User Expertise for Adaptive Interfaces. *Human-Computer Interaction*, 5(1):95–117, 1990.

[193] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[194] L. Vogel. *Eclipse 4 RCP: The complete guide to Eclipse application development*. Lars Vogel, 2013.

[195] W3C OWL Working Group. OWL 2 web ontology language document overview, 2009.

[196] J. Wang and J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *Proceedings of the 20th International Conference on Data Engineering*, pages 79–90. IEEE, 2004.

[197] Y. Wang, S. A. McIlraith, Y. Yu, and J. Mylopoulos. Monitoring and diagnosing software requirements. *Automated Software Engineering*, 16(1):3–35, 2009.

[198] T. Wehrmaker, G. Stefan, and K. Schneider. ConTexter Feedback System. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1459–1460. IEEE, 2012.

[199] I. H. Witten and E. Frank. *Data Mining*. Morgan Kaufmann Publishers, 2nd edition, 2005.

[200] F. Yetim, S. Draxler, G. Stevens, and V. Wulf. Fostering Continuous User Participation by Embedding a Communication Support Tool in User Interfaces. *AIS Transactions on Human-Computer Interaction*, 4(2):153–168, 2012.

[201] A. Zeller. *Why Programs Fail*. Morgan Kaufmann, 2nd edition, 2009.

[202] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[203] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.

[204] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. MIMIC: Locating and Understanding Bugs by Analyzing Mimicked Executions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 815–825. ACM, 2014.

List of Figures

List of Tables

Appendix A

Evaluation Material

This appendix contains material used in the empirical evaluation of the MALTASE framework described in Chapter 6.

A.1 Material for Evaluation of MALTASE-based Failure Reproduction

This appendix contains material used in the controlled experiment evaluating MALTASE-based failure reproduction described in Section 6.3. The material was adapted from Gurbanova [67].

Experimenters Guide

The following experimenters guide was used to help the experimenter conduct experiment sessions:

Before participant appears

1. Assign participant to group for 1st experiment randomly. The participant will be in the other group for the 2nd experiment. The next participant automatically goes to the other combination.
2. Choose order of tasks randomly.

With participant

3. Introduce participant to experiment (see Information Sheet)
4. Show MOSKitt Tutorial
5. If the group is experimental, introduce Timeline Tool:
   a. Introduce Timeline Tool by showing Timeline Tool video
   b. Let participant play with Timeline Tool
   c. Let participant answer first part of questionnaire "Understanding the Timeline Tool"

Experiment 1: Timeline Tool vs. Bug report

6. Start screen cast
7. Give participant the Timeline for 1st task (Experimental group)
   OR bug report for 1st task (Control group)
8. Let participant execute 1st task and observe.
   Stop task after 7 min.
9. Note down time taken and whether the failure could be reproduced.

10. Repeat steps 7.-9. with 2nd task.

Experiment 2: Timeline Tool + Bug report vs. Bug report

11. If the group is experimental, follow step 5.
12. Give participant the Timeline Tool + Bug Report for 3rd task (Experimental group)
    OR bug report for 3rd task (Control group)
13. Let participant execute 3rd task and observe.
    Stop task after 9 min.
14. Note down time taken and whether the failure could be reproduced.
15. Stop screen cast.
16. Ask participant to fill the questionnaire (see Questionnaire).
    Make sure to understand answers given to open questions.

After participant left

17. Complete notes and store screen cast.

Information Sheet

The following information sheet was used to help the experimenter introduce participants to the experiment:

Welcome to this experiment and thanks for participating!

The purpose of this work is to study how visualization of user interactions can help developers to reproduce failures.

Your identity in this study will be treated anonymously and no personal data will be given to anybody else. The results of the study may be published but we will not give your name or include any references to you.

The experiment will be conducted as follows:
You will have to reproduce failures with MOSKitt, an UML editor that will be introduced to you shortly. Reproduction means that you have to make failures happen which are described in textual bug reports or by the Timeline Tool. For some bugs you will get textual bug report and for others the Timeline Tool.

During the experiment, your interactions will be recorded.

After the tasks, we will ask you to fill a short questionnaire to give us your opinion about the Timeline Tool and failure reproduction with it.

Do not hesitate to ask questions if any!

Thank you again for your time!

Observer Sheet

The following observer sheet was used to help the experimenter record results and observations during experiment sessions:

Participant: ___
Date and Time: ___

Experiment 1: Bug report vs. Timeline Tool

Group (experimental, control): ___

Failure/ Bug Report 1: ___
Failure reproduced successfully? ___
Time needed: ___
Comments on Timeline usage: ___
Other comments: ___

Failure/ Bug Report 2: ___
Failure reproduced successfully? ___
Time needed: ___
Comments on Timeline usage: ___
Other comments: ___

Experiment 2: Bug report vs. Timeline Tool + Bug report

Group (experimental, control): ___

Failure/ Bug Report 3: ___
Failure reproduced successfully? ___
Time needed: ___
Comments on Timeline usage: ___
Other comments: ___

Questionnaire

The following questionnaire was used to collect participant feedback about MALTASE-based failure reproduction:

Understanding the Timeline Tool

Please answer the following questions while inspecting an interaction trace with the Timeline Tool:

1. What type of event occurred most frequently? ___
2. What was the last event before the first exception? ___
   What was the category of this event (application, user, environment)? ___
   When did this event occur? ___
   What type of node did the user create in this event? ___
3. What type of diagram was created in this trace? ___
4. What type of exception was thrown at the end of the trace and what is the error message of this exception? ___

Feedback on the Timeline Tool

Please indicate your honest opinion about the following statements (agreement on a 5-item Likert scale Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree):

- "The meaning of user interactions presented by the Timeline Tool is clear and easy to understand."
- "It is easy to find required information in the Timeline Tool."
- "Color helps to make a distinction between different types of events."
- "The GUI element titles (labels, tool tips, ...) are clear, concise and understandable."
- "The screen changes occurring upon my interactions (e.g. click on an event) are clear and seem reasonable."

Are there any parts of the Timeline Tool you found confusing or difficult to understand? Which ones? _ _ _

Which changes would you make to the Timeline Tool to improve its usability? _ _ _

Failure Reproduction with the Timeline Tool

Please indicate your honest opinion about following statements (agreement on a 5-item Likert scale Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree):

- "The Timeline Tool is helpful when reproducing failures."
- "It is clear what a user did when looking at the information provided by the Timeline Tool."
- "I prefer textual bug reports over visual Timeline Tool (given that both contain the same information)."

Which information did you miss in the Timeline Tool that would be helpful to reproduce failures? _ _ _

How can the Timeline Tool be improved to increase its support for failure reproduction? _ _ _

Personal Information (will be treated confidentially)

Gender: _ _ _

Age: _ _ _

Which of the following describes your current profession best (tick one)?
[ ] Student
[ ] Software Developer
[ ] Researcher

How do you judge your knowledge of UML (tick one)?
[ ] Beginner
[ ] Intermediate
[ ] Advanced

How long do you develop software (tick one)?
[ ] 0-1 year
[ ] 2-5 years
[ ] 5-10 years
[ ] > 10 years

How often do you usually reproduce failures in your daily work (tick one)?
[ ] Never
[ ] Once a month
[ ] Once a week
[ ] Daily

A.2 Material for Evaluation of MALTASE-based Skill Detection

This appendix contains material used in the case study evaluating MALTASE-based skill detection in Section 6.4. The material was adapted from Theiner [184].

Study Guide

The following study guide was used to help the researcher conduct study sessions:

Preparation
Before a study sessions starts, check the following points:
- Start the server.
- Start the client.
- The following documents should be on hand:
  - Info sheet
  - Questionnaire
  - The task sheet
  - The observer sheet
- Prepare the MOSKitt workspace:
  Create a new copy of the task handouts, archive old projects, and close all diagrams.
- Prepare MOSKitt video.

With participant
- Explain the purpose of the experiment.
- Hand the info sheet to the participant.
- Read the info sheet together and answer questions.
- Have the participant fill the questionnaire.
- Show MOSKitt introduction video.
- Ask for questions.
- Hand over the task sheet.
- Let participant work on the tasks.
- Observe the participant and record the time needed for each task.
- The third task can be solved in several ways and may take quite a while. However, after 15 minutes the task should be aborted, even if the user did not finish everything.

After participant left

- Check that interactions are saved in the database.
- Archive the MOSKitt workspace.
- Collect the sheets (questionnaire, observer sheet) and file them. Make sure that the participant id is noted on each of them.

Observer Sheet

The following observer sheet was used to help the researcher record results and observations during study sessions:

Date: ___
Participant: ___
Start time: ___

Task 1
Start: ___, End: ___
Observations: ___

Task 2
Start: ___, End: ___
Observations: ___

Task 3
Start: ___, End: ___
Observations: ___

End time: ___

Other observations: ___

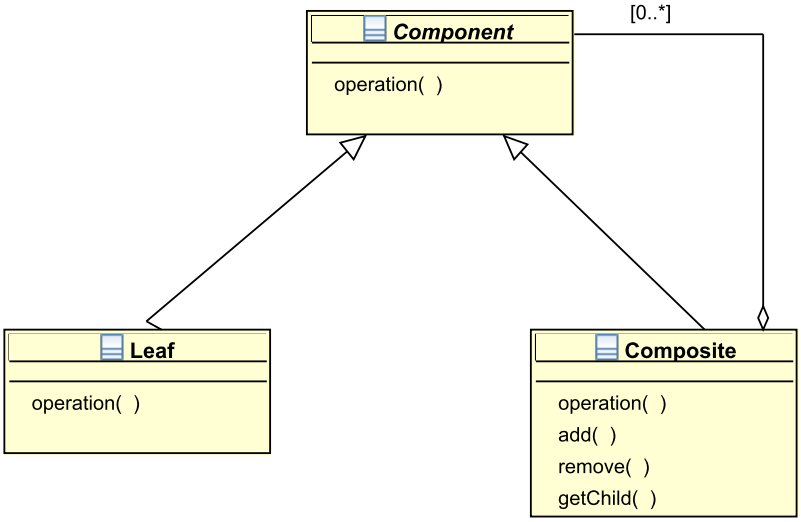
Task Descriptions

The following task descriptions were used in the evaluation case study. The exercises were adapted from Bruegge and Dutoit [23].

Task T1

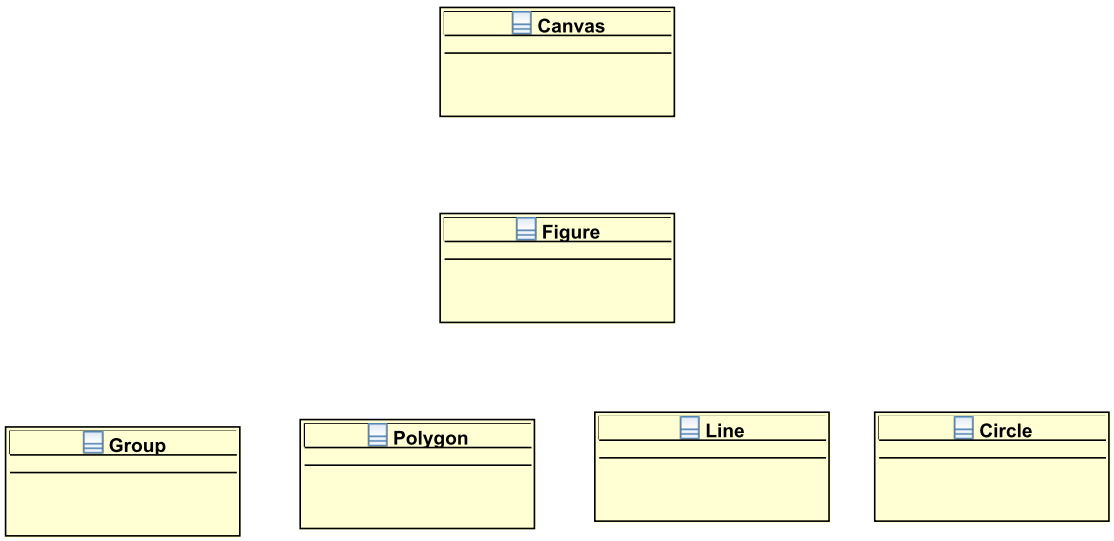
This task let's you get familiar with the basic features of MOSKitt.

- Open the existing UML class diagram "Task1" from the workspace.
- Delete any existing objects.
- Recreate the class diagram below in MOSKitt. Try to include all details like e.g. multiplicities.
- Save your work.



Task T2

Open the UML class diagram "Task2" from the workspace. The diagram should look like this:



The diagram is missing some properties and relationships between the classes. Please add the missing information of the following specification:

- A canvas consists of figures (Composition).
- A group consists of figures.
- A polygon consists of at least three lines.
- Figure is an abstract class with the property position and an abstract method draw().
- Group, Polygon, Line, and Circle are implementations (subclasses) of the class "Figure".

Save your work.

Task T3

Create a new UML class diagram named "Task3".

Create a UML model for the following situation:
In a small town there are five restaurants with a number of employees ranging from two in the smallest to 16 employees in the biggest restaurant. Each employee has an individual salary. There are two types of employees: cooks and waiters. Waiters wait on tables, pass the orders to the kitchen, serve the menus, and collect the money from the customers. The cooks prepare the menus. Each menu has its own price depending on the dishes, but each consists of at least a salad and a main course. Customers visit restaurants and order menus. The largest restaurant can serve up to 100 customers at a time while a restaurant might have no customers during bad times.

Save your work.

Questionnaire

The following questionnaire was used to collect participant information:

What is your gender?
[ ] Female
[ ] Male

What is your occupation?
[ ] Student
[ ] Researcher
[ ] Software developer
[ ] Other: ___

How do you rate your UML skills?
[ ] Never used UML
[ ] Beginner (Basic knowledge of the UML main capabilities)
[ ] Intermediate (Used UML for a few projects)
[ ] Advanced (Frequent use of UML or worked on large UML models)

For how long do you know UML?
[ ] Don't know UML
[ ] 0-2 years
[ ] 2-5 years
[ ] > 5 years

How often do you use UML for your work or study?
[ ] A few times a week
[ ] A few times a month
[ ] A few times a year
[ ] Never

How do you rate your MOSKitt skills?
[ ] Never used MOSKitt
[ ] Beginner (Started MOSKitt and tried it out)
[ ] Intermediate (Created a few simple UML diagrams in MOSKitt)
[ ] Advanced (Frequent use of MOSKitt or MOSKitt developer)

Have you worked with any other UML tool than MOSKitt?
[ ] No
[ ] Yes, with ___