



TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl für Angewandte Softwaretechnik



SCRIPT: A Framework for Scenario-Driven Prototyping

Harald Florian Stangl

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. Uwe Baumgarten
Prüfer der Dissertation: 1. Univ.-Prof. Bernd Brügge, Ph. D.
2. Univ.-Prof. Dr. Florian Matthes

Die Dissertation wurde am 31.05.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.06.2012 angenommen.

Acknowledgements

I would like to thank my advisor Prof. Bernd Brügge, Ph. D. for giving me the opportunity to research at his chair and write this dissertation. The discussions with him were always inspiring and brought new insights that I most probably would not have had otherwise. His cordiality made working at the chair an enjoyable experience and his affinity to industry provided me with real world project experience, for which I am very grateful.

I also want to thank Albert Feller and Johannes Lechner for the passionate discussions about the prototype model and for providing the tool support to deploy prototypes on the iOS platform, which was used in the experiment. Regarding the experiment, my thanks go to all the people who could take the time to participate.

I am grateful to all members of the chair for their support, especially Monika Markl, Helma Schneider, Uta Weber and Ruth Demmel on the administrative side, as well as all others who gave feedback on my research.

I want to thank my family, who continuously supported me from my early education to my final studies, as well as regarding all other aspects of my life.

Last, but most importantly, I am indebted to Helmut Naughton, who supported me from the very first moment that I started my research, always found time when I needed to discuss aspects of my work, and encouraged me to move on when I suffered a setback. Helmut, thank you very much.

Abstract

The fields of human-computer interaction and software engineering have evolved next to each other with different areas of concern. Software engineering focused on repeatable processes and reuse techniques to successfully deliver complex systems within time and budget in a changing environment. Human-computer interaction on the other hand focused on improving the interactions between users and systems and to deliver usable systems.

Some techniques have already found acceptance in both fields. Two of them are scenarios and prototypes. Scenarios describe concrete flows of interactions between a user and the system. In software engineering, scenarios are mostly used for requirements elicitation and system demonstrations during acceptance. In human-computer interaction, scenarios are used for usability tests and definition of the users' context to improve user acceptance of the delivered system. Prototypes are used by software engineers mainly to build early system demonstrations for technology evaluation and system integration tests. In human-computer interaction, prototypes are constructed to give users early hands-on experience and to evaluate different usage contexts.

The disadvantage of scenarios is that they do not themselves provide any interactivity. Although prototypes provide interactivity, their relation to the system requirements is often not clear. Using both scenarios and prototypes for system development forms a favorable combination, as each compensates for shortcomings of the other. But applying them simultaneously often results in artifacts that are hard to keep synchronized and consistent without a unified model.

This dissertation presents the SCRIPT framework for unified treatment of scenarios and prototypes, providing interactivity as well as traceability of requirements. Additionally, it allows to automatically generate videos that depict the interactions between user and system for demonstration purposes. The framework not only ensures that scenarios and prototypes are kept consistent, but also makes them accessible throughout development for activities such as use case specification and identification of analysis model elements. A controlled experiment was conducted, which shows that the SCRIPT framework ensures significantly better consistency of scenarios and prototypes compared with the traditional uncoupled usage.

Kurzfassung

Die Forschungsrichtungen Human-Computer-Interaction und Software Engineering haben sich parallel, mit unterschiedlichen Zielsetzungen entwickelt. Software Engineering hat sich auf wiederholbare Prozesse und Wiederverwendungstechniken fokussiert, die es erlauben, komplexe Systeme unter Einhaltung von Zeit- und Budgetvorgaben und unter sich ändernden Umständen zu entwickeln. Human-Computer Interaction dagegen hat sich zum primären Ziel gesetzt, die Interaktion zwischen Mensch und Maschine zu optimieren und benutzbare Systeme zu entwickeln.

Einige Techniken haben in beiden Forschungsrichtungen Akzeptanz gefunden, darunter auch Szenarien und Prototypen. Szenarien beschreiben konkrete Abläufe von Interaktionen zwischen einem Benutzer und einem System. Im Software Engineering werden Szenarien hauptsächlich zur Anforderungsermittlung und für Systemdemonstrationen bei der Kundenabnahme eingesetzt. Human-Computer-Interaction verwendet Szenarien für Usability-Tests und zur Definition des Kontextes, in dem ein System verwendet wird, um die Akzeptanz des gelieferten Systems zu erhöhen. Prototypen hingegen werden von Softwareentwicklern primär für Systemdemonstrationen zur Technologieevaluierung eingesetzt, sowie zur Durchführung von Systemintegrationstests. Human-Computer-Interaction verwendet Prototypen, um Anwendern schon früh ein Gefühl für das System zu vermitteln, und um unterschiedliche Anwendungsbedingungen zu evaluieren.

Ein Nachteil von Szenarien ist ihre fehlende Interaktivität. Obwohl Prototypen diese Interaktivität mit sich bringen, fehlt ihnen meist der explizite Bezug zu den Systemanforderungen. Daher stellt die Verwendung sowohl von Szenarien wie auch Prototypen in der Systementwicklung durch ihre gegenseitige Ergänzung eine vorteilhafte Kombination dar. Allerdings führt die gleichzeitige Verwendung von Szenarien und Prototypen zu Problemen bei der Synchronität und Konsistenz, wenn ihnen kein gemeinsames Modell zu Grunde gelegt wird.

Diese Dissertation präsentiert das SCRIPT Framework, das eine gemeinsame Basis für Szenarien und Prototypen bildet und sowohl Interaktivität wie auch eine Verknüpfung mit Anforderungen ermöglicht. Es erlaubt ausserdem die automatische Erzeugung von Videos, die die Interaktion zwischen einem Benutzer und dem System zeigen, und die für Diskussionszwecke genutzt werden können. Das Framework stellt nicht nur sicher, dass Szenarien und Prototypen konsistent gehalten werden, sondern erlaubt auch ihre Verwendung für weitere Entwicklungsaktivitäten, wie die Spezifikation von Use Cases

und die Identifizierung von Analysemodellelementen. Mit Hilfe eines kontrollierten Experiments konnte gezeigt werden, dass das SCRIPT Framework eine signifikant bessere Konsistenz zwischen Szenarien und Prototypen sicher stellt, als es mit der bisherigen, unverknüpften Verwendung der Fall war.

Contents

Acknowledgements	iii
Abstract	v
Kurzfassung	vii
Conventions	xiii
1 Introduction	1
1.1 The Problem	3
1.2 Outline	5
2 User-Centered Software Engineering	7
2.1 Human-Computer Interaction	8
2.2 Object-Oriented Software Engineering	9
2.3 Mental Models and System Models	11
2.4 Approaches for User-Centered Software Engineering	13
2.4.1 Approaches with Focus on Processes	14
2.4.2 Approaches with Focus on Artifacts	15
3 Requirements	19
3.1 Requirements Visualization and Simulation	20
3.2 Use Case Visualization	21
3.3 Scenarios	22
3.3.1 Scenario Formalization	23
3.3.2 Scenario Visualization	25
4 Prototyping	27
4.1 Prototyping vs. Design	28
4.2 Prototypes as Artifacts	30
4.2.1 Categories of prototypes	31
4.2.2 Horizontal and Vertical Prototypes	32
4.2.3 Prototype Fidelity	32
4.2.4 Focus of Prototypes	34
4.3 Prototyping as Process	35

4.3.1	Revolutionary prototyping	35
4.3.2	Experimental Prototyping	36
4.3.3	Evolutionary Prototyping	36
4.4	Techniques for Prototyping	36
4.4.1	Wireframe Prototyping	37
4.4.2	Storyboard Prototyping	37
4.4.3	Paper Prototyping	39
4.4.4	Digital Prototyping	39
4.4.5	Video Prototyping	40
4.4.6	Wizard-of-Oz Prototyping	41
4.5	Prototyping Tools	41
4.6	Prototype Knowledge Management	43
5	The SCRIPT Model	45
5.1	Scenario Meta Model and Interaction Meta Model	48
5.2	Scenario Prototype Structural Meta Model	51
5.3	Scenario Prototype Interaction Meta Model	55
5.4	Relationship between Scenario and Scenario Prototype	58
5.5	Criteria of Applicability	59
5.5.1	Platforms	59
5.5.2	Modes of Interaction	60
5.5.3	Degree of User Interface Content Change	61
5.5.4	Amount of User-System Interaction	61
6	Application of SCRIPT	63
6.1	Activities in the SCRIPT Framework	63
6.2	SCRIPT in Development Lifecycles	65
6.3	Sequence of Model Traversal	65
6.4	Graphical Input for SCRIPT	66
6.4.1	Paper-Based Sketching	67
6.4.2	Digital Sketching	67
6.4.3	Building from Predefined Shapes	68
6.5	System Specification	68
6.5.1	Deriving Use Cases	69
6.5.2	Extracting User Interface Model	70
6.5.3	Identifying Analysis Model Elements	71
6.6	Document Export	75
6.6.1	Static Documents	75
6.6.2	Video Generation	76
7	The SCRIPT Editor	79
7.1	User Interface	79

7.1.1 Scenario Prototype Editor	80
7.1.2 Scenario Editor	84
7.2 Architecture	85
7.3 Components of Evaluation Setup	86
8 Evaluation	89
8.1 Experimental Design	89
8.2 Tasks	91
8.3 Experiment Results	93
8.3.1 Number of Errors	93
8.3.2 Working Time	94
8.3.3 Exit Interview	94
8.3.4 Threats to Validity	95
9 Conclusion and Future Work	99
9.1 Contributions	99
9.2 Future Work	100
A Experiment Prototypes Description	103
A.1 Storyboard for Scenario 1: “Peter comes home”	104
A.2 Storyboard for Scenario 2: “Peter programs”	108
B Experiment Work Package Descriptions	113
List of Figures	129
Bibliography	131

Conventions

In this dissertation, the following conventions are used:

- Citations are given in alpha style, e.g. [ABCD12], where each letter stands for the initial letter of the authors' surnames (e.g. authors Alpha, Beta, Charlie and Delta), followed by the last two digits of the year of publication. Whenever there are more than four authors, the fourth and all additional authors are indicated by a "+" character, e.g. [ABC+12]. For publications with only one author, the first three letters of their surname is given, e.g. [Abc12].
- Definitions of terms that are used throughout this dissertation are printed in **bold** face.
- Terms that have been coined by other authors are printed in *italics*.
- Terms relating to the realization of the SCRIPT framework are set in **sans-serif** font whenever they are introduced first.
- All diagrams in this dissertation are based on the Unified Modeling Language (UML), if not stated otherwise.
- In order to ensure gender neutrality, the "singular they" form is used whenever appropriate.

Trademark notice: Product names or corporate names that appear in this dissertation may be (potentially registered) trademarks, they are used for referencing purposes only without intent to infringe.

Chapter 1

Introduction

Over the last years, the amount of mobile devices that are in everyday use, like smartphones and tablet computers, raised significantly. One of the reasons for their success are the openly accessible development and distribution platforms that allow developers from all over the world to develop and distribute software for these devices. With the increasing computational power and the availability of sensors that are embedded, these mobile devices can provide functionality that formerly required dedicated devices. Users no longer need to carry one device per functionality, e.g. for making telephone calls, listening to music, making photos, or navigating. Instead, they only need to have a smartphone and one application per desired functionality. Additionally, new interaction paradigms such as touchscreens for input and output at the same time and allowing users to interact with their device using finger gestures helped to increase the acceptance—and demand—of these devices. In the last quarter of 2011, Apple sold over 37 million iPhones and 15 million iPads [App12a]. Apple’s competitors such as Samsung, HP and Amazon are following this trend with their own tablet computers, mostly employing Google’s Android operating system. In total, they sold more than 10 million devices in the same time range. Compared to the last quarter of 2010, smartphone sales increased by 58% [Gar12] and tablet sales increased by about 150% in the last quarter of 2011 [Ana12].

Mobile computing devices pose challenges for the software developer, especially regarding the design of the user interface. Established user interface paradigms such as window–icon–menu–pointer (WIMP) are only partially applicable [SY06, Li09], mostly due to reduced screen sizes and novel input modalities. Several additional aspects need to be taken into account when designing user interfaces for the mobile user. Whenever users sit in front of their “desktop” computer, no matter if it is a stationary computer or a laptop, their primary focus is on the interaction with the computer. In contrast, interactions with a mobile device often take place in situations where the primary focus of a user is not on the interaction with the device, but on a different task and in a changing context. For example, doctors making their rounds in a hospital might use a tablet computer to retrieve information about the patient they currently examine. The task of the doctors that receives the primary focus is to see after their patients, while the use of the tablet computer only serves to support this task. In order to build software

that effectively fulfills its supporting role, the **context** of the user such as the environment, e.g. ambient noise and lighting conditions, current social setting of the user, e.g. being alone or in a group of people, and technical details such as network availability needs to be taken into account [GM03, JL08]. The context is subject to change while the software is in use, for example, when a doctor changes the room on their round, the lights in the room they enter might be turned off so that the screen is no longer readable, the wireless network connection might break down, or voice input might no longer be possible because the television is running.

Usability describes how usable a system is for the user. This dissertation follows the definition of usability by Nielsen [Nie93], who defines five aspects of usability: *Learnability* describes how easy it is for a user to learn how to use a system; *Efficiency* describes how fast users can operate a system once they learned how to use it; *Memorability* describes how much effort is needed for users to re-learn how a system is used when they did not use it for a longer period of time; *Errors* describes how many errors users make, how severe the errors are and how easy users can recover from errors they made; *Satisfaction* describes how pleased users are with using a system.

Providing good usability on mobile devices is not only a noble aim to strive for, but has significant financial consequences, as a study of the mobile phone market in the UK shows [Ove06]. The study reports that one out of seven mobile phones that were sold has been returned in the first year after purchase. From the returned phones, about 63% had no hardware or software fault, but instead users complained about usability problems, configuration problems or mismatched expectations. For the manufacturers, the returns summed up to costs of 54 million British Pounds and, if extrapolated to worldwide mobile phone sells, to as much as nearly 4.5 billion U.S. Dollars.

In contrast, the amount of available applications for users of mobile devices to choose from is growing rapidly. Apple's AppStore currently offers more than 550.000 apps and more than 25 billion downloads have been recorded since the opening of the AppStore in July 2008 [App12b]. Google Play, which recently replaced the Android Market, offers over 450.000 apps and more than 13 billion downloads have been counted since October 2008 [Lie12]. More often than not, multiple alternative applications exist that can be used to solve a task at hand. When given the choice, most users can be expected to decide for the application that not only provides the required functionality, but which also provides the best usability.

For developers, the rapidly increasing amount of mobile devices and their special needs regarding usability, as well as the keen competition on the software market and the financial effects of bad usability make a strong orientation towards the users of a system necessary. The problem of developing user interfaces that provide the best possible usability—no matter if for mobile devices or any other system—is one of the main focuses of the field of human-computer interaction (HCI). The increasing computational power of mobile devices, however, allows to run sophisticated software that requires development efforts similar to traditional desktop applications. When it comes to develop complex and large-scale systems, the field of software engineering (SE) takes an important role

with its repeatable processes and reuse techniques that help to deliver high quality systems within time and budget in a changing environment.

Unfortunately, the two disciplines evolved next to each other without major cross-pollination, as has been discussed in multiple conference workshops [KBB03, HVF03, JBKC04]. Given the increasing attention mobile devices receive both for personal and professional use, it is more important than ever that HCI and SE come together in order to develop approaches that excel in solving technical challenges and providing the best possible usability.

1.1 The Problem

Some techniques have found acceptance in both HCI and SE and may help to close the gap between them. Two of them are scenarios and prototypes, which are in the focus of this dissertation.

A **scenario** defines a single, concrete flow of interactions between a user and the system, where the user tries to solve some specific task with help of the system. An **interaction** can be initiated by the user, such as pressing a button, or by the system, such as playing an alarm sound. A scenario not only defines the interactions, but it may also describe the context in which the interactions take place. Scenarios can be represented in different ways, for example as **narrative scenarios** using natural language, which can contain inconsistencies and ambiguities, or using a formal language to ensure that the representation is consistent and unambiguous. Independent of their representation, however, scenarios should not contain technical details about the system, but only use terminology from the application domain.

In software engineering, scenarios are mostly used for requirements elicitation and system demonstrations during client acceptance tests, where the term **client** describes the role of the person who awarded the project contract and who is mostly interested in managerial and financial issues regarding the project. In human-computer interaction, scenarios are used for usability tests and definition of the users' context to improve user acceptance of the delivered system, where the term **user** describes the role of the people who are eventually going to work with the system once it has been deployed.

When scenarios are represented as narrative scenarios, they have both benefits and drawbacks. Their narrative style and use of terminology solely from the application domain makes it possible for all stakeholders to take part in the requirements elicitation process, as the narrative representation does not require any knowledge about formal notations. In this dissertation, the term **stakeholder** stands for both clients and users. The lack of an underlying structure, however, is one of the drawbacks of using narrative scenarios for describing requirements, as the information it contains cannot easily be accessed without reading the whole scenario. It is also not possible to establish a relationship between single parts of a narrative scenario and subsequent development artifacts like use cases or functional requirements. Another disadvantage of narrative

scenarios is their lack of **interactivity**, which is the capability of an artifact to react to user input. While a narrative scenario describes the interactions between a user and a system in detail, it does not provide any way for users to actually experience and evaluate the interactions themselves.

A **prototype** is a system that contains selected functionality of a larger system under development, potentially also with reduced quality compared to the requirements of the system under development, such as missing security constraints or lower performance. Prototypes are used by software engineers mainly to build early system demonstrations for technology evaluation and system integration tests. In human-computer interaction, prototypes are constructed to give users early hands-on experience and to evaluate different usage contexts. Prototypes can help in the evaluation of the feasibility of technical solutions like algorithms or architectures, or for the evaluation of user interface design proposals by letting stakeholders experience the flow of interactions prior to system implementation. This way, problems can be identified and resolved early on, reducing costly changes that would otherwise only be detected during later stages of development or even after deployment. However, prototypes fall short regarding two aspects. First, they only convey an idea about how interactions between user and system are supposed to take place, but they do not provide any information about the context of system use. Depending on the type of system that is being developed, information about the context of system use can be essential for making decisions during later phases of software development. Regarding the example of doctors making their rounds as described above, the information that the wireless network connection might break down when a doctor changes the room can guide developers to use appropriate design patterns to cope with this situation. Second, prototypes are often not well documented, so that only the creator of a prototype has information about which functionalities of a system have been prototyped. If this person is no longer available and the information has not been externalized so that it is accessible by other project members, the utility of a prototype can be reduced significantly [Sch96].

Using both narrative scenarios and prototypes can help in mitigating the problems mentioned above. Narrative scenarios provide information about the context of system use that cannot be derived from a prototype, and prototypes provide the interactivity that is lacking in narrative scenarios. But if narrative scenarios and prototypes are developed in parallel without an underlying unified model, there is the risk that the resulting artifacts diverge and thus lead to an inconsistent requirements specification (see also Chapter 8).

To overcome these problems, this dissertation presents the SCRIPT framework. It provides the SCRIPT model, which is a unified meta model for modeling narrative scenarios and prototypes, making them accessible for subsequent development steps on a fine-grained level, e.g. by referencing individual interactions of a scenario in a use case step, or individual user interface elements of a prototype when identifying analysis model elements. The meta model establishes a close relationship between narrative scenarios and prototypes, so that each prototype is related to a narrative scenario that

describes the interactions that are realized by the prototype, including the context of its application. Each narrative scenario is accompanied by a prototype that allows users to experience the interactions themselves that are described in the narrative scenario. This close relationship allows to perform continuous consistency checking to ensure that narrative scenarios and prototypes stay consistent when either of them is changed. The SCRIPT Editor, which is the reference implementation of the SCRIPT model, utilizes this relationship to automatically notify the developer whenever a change to a narrative scenario requires changes to the according prototype in order to keep them consistent, and vice-versa. Additionally, this dissertation describes how models that are based on the SCRIPT model can be used to automatically generate different scenario representations, including storyboards (see Section 4.4.2) and videos that depict the interactions between user and system for demonstration purposes.

A controlled experiment was conducted, which shows that the SCRIPT framework ensures significantly better consistency of narrative scenarios and prototypes compared to uncoupled usage.

1.2 Outline

The remainder of this dissertation is organized as follows: Chapter 2 gives a short introduction to the fields of human-computer interaction (HCI) and software engineering (SE) and presents existing approaches that try to close the gap between the two fields. These approaches are categorized depending on whether they focus on processes or artifacts, where the approach presented in this dissertation belongs to the latter category. Chapter 3 focuses on scenarios and use cases as means for gathering requirements, and presents approaches for either representing them using formal languages or making them better understandable by enhancing them with appropriate visualizations. Chapter 4 gives an overview of the field of prototyping and prototype classifications, and presents approaches for prototyping as well as for managing knowledge that is contained in prototypes.

Chapter 5 describes the SCRIPT model, which consists of the scenario meta model and the interaction meta model that allow for the definition of interactions between user and system in an implementation-independent way, and the scenario prototype structural meta model and scenario prototype interaction meta model that allow for the creation of prototypes for two-dimensional graphical user interfaces. Chapter 6 explains the activities that belong to the application of the SCRIPT framework in a software development project, and how artifacts that result from the application of the SCRIPT framework can be utilized for subsequent development steps besides the modeling of scenarios and prototypes. Chapter 7 gives an overview of the user interface and the architecture of the SCRIPT Editor, which has been developed in order to evaluate the SCRIPT framework. Chapter 8 presents the experiment that has been conducted to

show that the SCRIPT framework improves the consistency of narrative scenarios and prototypes when they are developed in parallel, and discusses its results.

Chapter 9 summarizes the contributions that have been presented in this dissertation and gives an outlook to future directions of research.

Chapter 2

User-Centered Software Engineering

The disciplines of human-computer interaction (HCI) and software engineering (SE) have largely developed next to each other, each with different areas of concern. Software engineering focused on the development of repeatable processes, identification of principles, and reuse techniques to successfully deliver complex systems within time and budget in a changing environment, while human-computer interaction focused on studying and improving the interactions between users and systems and to deliver usable systems. Although it seems that the two fields would benefit from each other, their integration has not yet happened in the large.

Seffah et al. [SGD05] observed that the challenges in the collaboration of practitioners from HCI and SE are manifold. First, people from both disciplines have different backgrounds and different understandings of usability (*people gap*) and both regard their respective fields as the more important (*responsibility gap*). On the technical side, the increased decomposition of systems into manageable parts leads to a strong decoupling of user interface and underlying system. While this is a desired outcome in general, it poses a problem to improving usability of systems, as this often not only requires changes to the visual parts of a user interface alone, but also involves fundamental changes to the flow of interactions that are supported by a system (*modularity fallacy*).

From a management perspective, managers often have the impression that they cannot spend resources on usability, which also results in a lack of training for developers. In contrast, Marcus describes some examples and figures that show how good usability can also be quantified in terms of return-on-investment (ROI) [Mar02]. Nielsen advocates the *discount usability engineering* approach [Nie93], where **usability engineering** describes a sub-discipline of HCI that is devoted to the development of methods for increasing the usability of software systems. Discount usability engineering only requires the use of three techniques that are both easy to learn and cheap to apply: *scenarios*, *simplified thinking aloud* and *heuristic evaluation*. Nielsen's definition of *scenarios* actually refers to prototypes that allow users to play through a single scenario, where the sequence of user actions and system reactions is predetermined, and which are thus simple to create. *Simplified thinking aloud* is a method where users explain to developers their thoughts and experiences while they play through a scenario, but where developers only take notes instead of the organizational overhead of videotaping the whole session and

analyzing it afterwards, as it is the case with traditional thinking aloud user studies. With *heuristic evaluation*, Nielsen promotes the application of a small set of usability heuristics instead of creating large style guides with thousands of rules developers are required to learn and follow.

The chapter is organized as follows: Section 2.1 gives a brief overview of the roots and sub-disciplines of HCI, especially “user-centered design”. Section 2.2 describes the principles of object-oriented software engineering and the software development methodology this dissertation is based on. Section 2.3 describes the mental models of the designer and the user of a system, and how they relate to models of the system. Section 2.4 presents approaches that integrate HCI and SE into a discipline of user-centered software engineering, either by coupling processes from both fields, or by defining artifacts that serve as a bridge between both fields.

2.1 Human-Computer Interaction

Already back in 1990, Chignell stated “that there is as yet no science of human-computer interaction” [Chi90]. He proposed a taxonomy that should serve as a guideline along which HCI could evolve. The top-level elements of the taxonomy that he defined were *basic interface model*, *cognitive engineering*, *user interface engineering* and *applications*. The *basic interface model* is concerned with interactions between users and systems, *cognitive engineering* consolidates approaches for applying models from cognitive science to user interface design, *user interface engineering* collects the details about concrete user interface design and guidelines, and *applications* gives a drafted sub taxonomy of application domains which, to his consideration, are in need for HCI.

According to the ACM SIGCHI Curricula for Human-Computer Interaction [HBC⁺96], the field of HCI has its roots in the fields of “computer graphics, operating systems, human factors, ergonomics, industrial engineering, cognitive psychology, and the systems part of computer science”. Hewett et al. [HBC⁺96] provide a working definition for HCI, as they constitute that no commonly agreed upon definition for HCI exists yet: “Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.”

The definition names “design [...] of interactive computing systems” as one of the interests of HCI. In a more general context, Norman [Nor02] defined **user-centered design** as a methodology for product design, where the needs and interests of the user are the most important aspect and have priority over other aspects such as wealth of functionality or aesthetics. Norman’s definition refers to the design of physical products as well as software systems. Karat defined four main principles for user-centered design of computer systems: early and continuous focus on users, early and continuous evaluation, iterative design and integrated design [Kar97]. He puts a special focus on the fact that usability is not solely decided by the design of the user interface, but also depends on

how well a system fits into its context of use. Based on the same idea, Gulliksen et al. developed a set of design principles for user-centered design [GGB⁺05]. They derived and refined it from their experience of an in-house study in their company, where they started with an initial set of principles and modified them as the project into which the study was embedded proceeded. As part of the principles they compiled, they advocate the use of prototypes and evaluation of the planned system interactions in the real context of the user. While this evaluation could be done with paper sketches and static images of the user interface in the beginning, prototypes should be employed in later stages of development.

In 2010, the International Organization for Standardization (ISO) released the standard 9241-210 “Human-centred design for interactive systems” [ISO10], which replaced the standard ISO 13407 on human-centred design that was published in 2000. The standard uses the term **human-centred design** instead of user-centered design to make explicit that it focuses not only on the people directly using a system, the users, but also all other people who are indirectly involved or affected by the use of a system. According to ISO 9241-210, human-centred design is an “approach to systems design and development that aims to make interactive systems more usable by focusing on the use of the system and applying human factors/ergonomics and usability knowledge and techniques”. Any human-centred design approach should comply with the following six principles [ISO10]: “the design is based upon an explicit understanding of users, tasks and environments; users are involved throughout design and development; the design is driven and refined by user-centred evaluation; the process is iterative; the design addresses the whole user experience; the design team includes multidisciplinary skills and perspectives”.

2.2 Object-Oriented Software Engineering

Object-oriented software engineering is based on the idea of object-oriented modeling and design [RBP⁺91]. A **model** is an abstraction of entities or concepts. As it is an abstraction, a model leaves out details of the entity or concept it describes in order to make it easier to understand and to focus on relevant aspects only. This requires a model to have a purpose, as the purpose decides which are the relevant aspects that should be part of the model. For the same entity or concept, it is possible to have a multitude of models, all with different purposes and thus leaving out and representing different aspects.

In object-oriented modeling, the basic building blocks of models are objects. An **object** represents a single, unique entity or concept and encapsulates both data and behavior. Rumbaugh et al. [RBP⁺91] define four attributes that are common for object-orientated approaches: *identity*, *classification*, *polymorphism*, and *inheritance*. First, two objects, even if they contain exactly the same data and behave the same way, are

treated as two distinct objects (*identity*). Second, objects that share the same data structure and behavior are grouped into **classes**. Objects that belong to the same class are called instances of that class (*classification*). Third, the same operation may behave differently when it is invoked on different classes, while the selection of the correct behavior is hidden from the caller of the operation (*polymorphism*). Fourth, classes can be organized hierarchically, where classes in a lower position of the hierarchy inherit the data structure and behavior of their parent classes and augment it with additional data structures and behavior (*inheritance*).

The idea of object-oriented software engineering then is to use object-oriented models for building a model of the software to develop. Development starts with building models of the application domain, without any reference to technical details about how the software should be built. These application domain models then get gradually refined with details about the system to be developed, i.e. the models of the application domain are enhanced with concepts from the solution domain. When the models are sufficiently detailed, they are used as a basis for implementation.

Software engineering is concerned with both the technical challenges of developing software, as well as with the managerial aspects that need to be considered in order to successfully deliver working software. The IEEE “Standard for Developing Software Life Cycle Processes” (IEEE Std 1074-1997) [IEE97] defines 65 activities that need to be taken into account when planning for a software development project, of which 17 belong to the project management group.

For this dissertation, the approach for object-oriented software engineering as described by Bruegge and Dutoit is taken as reference [BD09]. They define three types of models: the *functional model*, the *object model* and the *dynamic model*. The *functional model* describes the functionality of the system from the point of view of the user, the *object model* describes the structure of the application domain and the system to be built, and the *dynamic model* describes the interactions between user and system as well as system-internal behavior.

Their development process is based on the IEEE standard 1074-1997, but follows a object-oriented methodology. As the challenges of project management are not of primary interest in this dissertation, only the six top-level activities of software development are described in more detail. These are *requirements elicitation*, *analysis*, *system design*, *object design*, *implementation* and *testing*. During *requirements elicitation*, the goal is to identify the purpose of a system, including a description of the context in which it is going to be used. In the *analysis* activity, a model of the application domain as well as of the system to be built is created, although still on an abstract level without any reference to concrete technologies used for realization. A first step towards realization is taken during *system design*. In this activity, developers decompose the system into smaller parts, define the hardware the system is supposed to run on and decide on other high-level realization issues. The *object design* activity focuses on the creation of low-level models of the system to be built, use of design patterns and concrete subsystem interface design. *Implementation* refers to the activity of actually programming the sys-

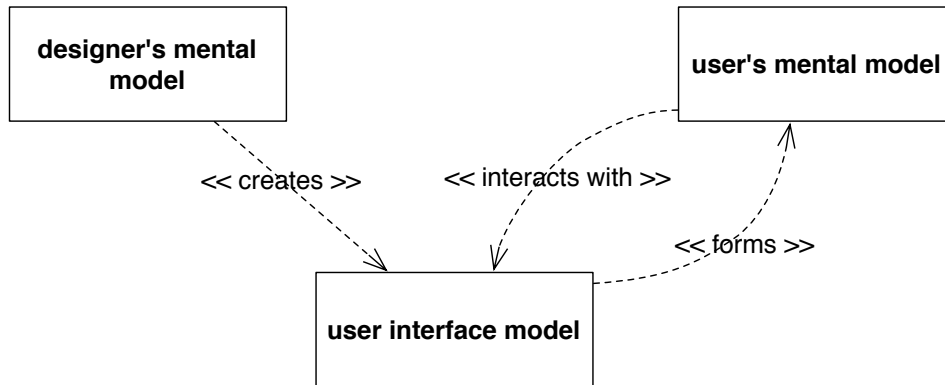


Figure 2.1: Mental models of designer and user. Adapted from [Nor02].

tem, and *testing* ensures that all parts of the system work together and that the system fulfills the expectations of the clients and users.

Note that interface design, which is part of object design, refers to computer–computer interfaces as well as interfaces between user and system. The design of the user interface, however, already starts during requirements elicitation, where hand drawn sketches or static images of the user interface are used to support the identification of requirements. During analysis, boundary objects are defined that represent the interface between user and system on an abstract level, for example a form that is presented to a user, but without details about the arrangement of elements that are placed on the form. Additionally, navigational paths are defined that describe how users can navigate between parts of the user interface. The detailed design of the user interface is then part of the object design activity.

The order in which the activities are presented here should not imply any temporal relationship. Only the development lifecycle that has been selected for a project decides if, when and how often each activity is to be executed during the development of a system.

This dissertation is focused on the activity of requirements elicitation, which is typically one of the first activities of a software development project. Achieving a complete, consistent and correct requirements specification as a result from this activity is crucial, as subsequent development steps rely on it. This means that errors or omissions that have been introduced during requirement elicitation will need to be fixed later on, which results in increased costs compared to having it done right in the first place [Boe81].

2.3 Mental Models and System Models

In order to better understand how users interact with a system, Norman [Nor02] describes the mental models of the designer and the user, and how designer and user communicate by means of the user interface, which is shown in Figure 2.1. When a

designer starts to work on the development of a system, they form a mental model of how a user is supposed to interact with the system. This mental model of the designer is represented by the *designer's mental model*. The designer then transforms their mental model into the *user interface model*, which represents the user interface that is eventually shown to the user. Upon interaction with the user interface and its underlying model, the user forms a mental model about how they think the system actually works, which is represented by the *user's mental model*. In many cases, like when developing a product for mass distribution, designer and user do not get in contact with each other directly, but only via the user interface model. Ideally, the mental model that the user develops is identical or at least very close to the designer's mental model, meaning that the user understood how the designer intended the system to be used. Depending on the system, however, users can be able to operate it even when their mental model is completely different from the designer's mental model, although in these cases the probability of a user making an error is much higher compared to if they had the same mental model as the designer [Nor02].

In the context of software engineering, the user interface model represents only one part of the complete system model. As described in the previous section, the system can be modeled from various viewpoints with help of the functional model, the dynamic model and the object model, which is shown in Figure 2.2. The models depicted here are not disjoint, however, as the user interface model contains aspects of all three system models. It is part of the functional model, as the user interface plays an important role in defining how the user can access functionality of the system; it is part of the dynamic model, as the flow of interactions between user and system need to be defined; and it is part of the object model, as the components that make up the user interface are described in terms of objects and their relations.

The approach presented in this dissertation, with its focus on scenarios and prototypes, also contributes to all three system models. Scenarios participate in the functional model, as they give explicit examples of how a user accesses functionality of the system. They also participate in the dynamic model, as they describe how the flow of interactions between user and system take place. Prototypes participate in the object model, especially of the user interface, as they provide a preliminary version of the user interface for users to interact with. They also participate in the dynamic model, where the interactions that are realized by the prototype are defined. These interactions are supposed to be the same interactions that have been defined by the scenarios, so that scenarios and prototypes actually represent the same set of interactions. The meta models for scenarios and prototypes, which are part of the SCRIPT framework, ensure that this is the case, so that scenarios and prototypes are kept consistent.

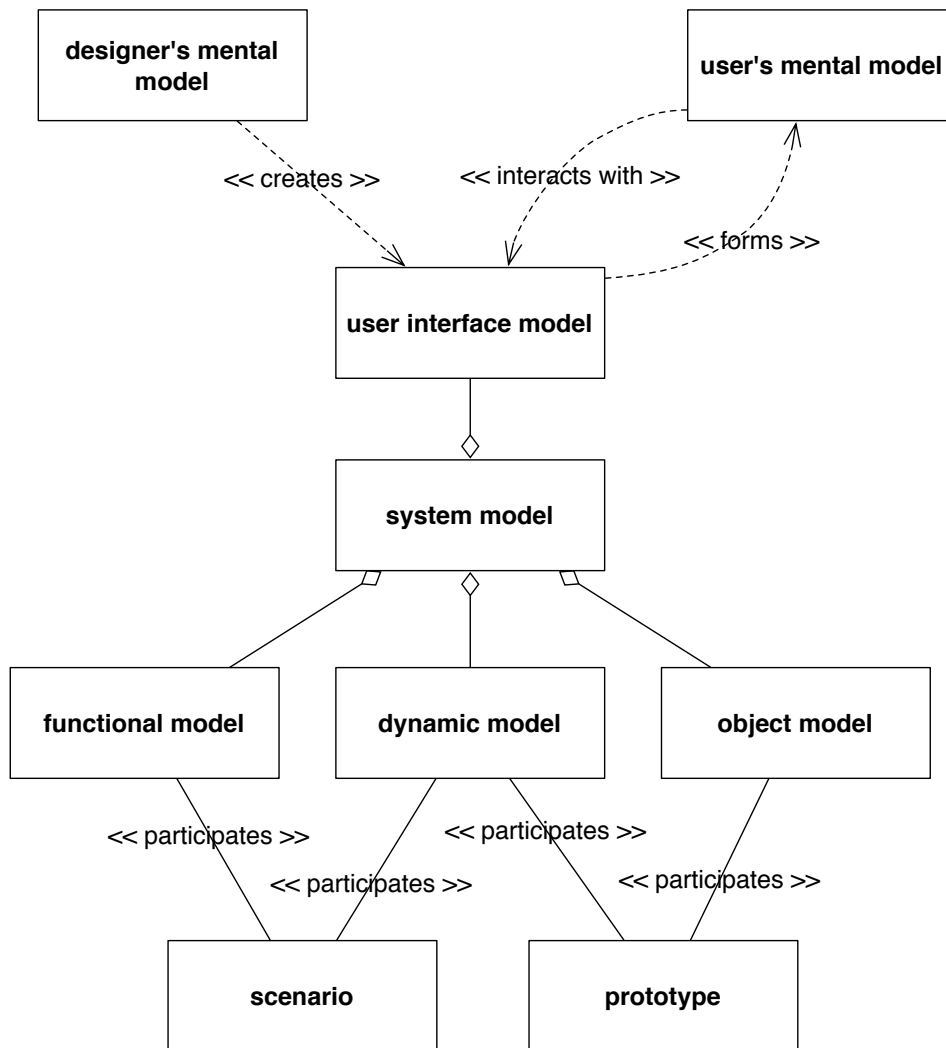


Figure 2.2: Mental models and system models

2.4 Approaches for User-Centered Software Engineering

The International Organization for Standardization realized the importance of usability in software development, which led to the amendment of ISO standard 12207:1995 for software processes by a usability process in Amendment 1. This amendment has been integrated into the main standard in ISO 12207:2008.

The adoption to everyday work practice, however, has not yet happened. A survey with 63 HCI and 33 SE professionals that has been conducted by Jerome and Kazman [JK05] shows that collaboration between practitioners from HCI and SE is basically not happening. People working in one field have barely any knowledge about the other

field, there is only a minimum of working together, and if, contact between HCI and SE typically only happens late in development, during implementation and testing, which makes it expensive to compensate for problems that have been introduced in the very beginning of a project .

Workshops have been held in order to figure out how SE and HCI can work together [BPTM03, JBKC04]. Also Seffah et al. investigated possible ways for integrating HCI and SE [SM04, SDM05].

In the following, approaches that try to advance user-centered software engineering by combining techniques and artifacts from HCI and SE are presented, first with a focus on the process perspective, followed by approaches that focus on the artifacts that are used by both disciplines. The approach presented in this dissertation with its focus on scenarios and prototypes also belongs to the latter category.

2.4.1 Approaches with Focus on Processes

In order to make software engineers familiar with usability, Jokela proposes a workshop approach called *KESU URD* (usability requirements development), where usability professionals and software engineers work together in a series of workshops in order to determine the usability requirements for a project [Jok05]. The workshop setup defines certain roles: a facilitator, who is a usability specialist and not part of the project team; a member of the project team, who is responsible for usability; a decision maker, typically the project manager; and a group of so-called “analysts” that consists of developers, but may also contain people from other teams such as technical documentation and customer service. The workshop consists of 10 steps that lead to the definition of a *usability requirements table* in the end. An additional benefit of the collaborative structure of the workshop is that the participants also learn from the usability specialist while they work on defining the usability requirements.

The challenge of augmenting existing software development processes with usability techniques has been tackled by Ferre et al. [FJM05] First, they define a relationship between usability techniques and software development activities. Then they give a description about which usability techniques can be applied when in a software development process. They do not focus on a specific process, but only require it to be iterative. In order to be applicable to as many existing software development processes as possible, they define a generic iterative process and explain when to apply which usability technique referring to this generic process description. By mapping the concrete software development process to the generic process, a project manager can decide which usability techniques should be used.

Blomkvist analyzed the relationship between user-centered design processes and agile development [Blo05]. He found out that although both share some basic principles like iterative development and focus on people instead of processes, both are lacking certain aspects with regard to the other: agile development does not necessarily put the user in the center of development, but rather focuses on details about how a system

is developed, and user-centered design processes do not cope well with frequent change in requirements. He proposes three ways of overcoming these limitations: either by enriching agile development with user-centered design methods, by making user-centered design more agile, or by merging both disciplines into a combined approach.

An attempt to combine eXtreme programming (XP) [BA04] and usability engineering (UE) is *eXtreme usability (XU)* presented by Holzinger and Slany [HS06]. It aims to bring usability engineering techniques to eXtreme programming and thus merge both. While XU is still in development, first evaluations with students showed that it is a viable approach.

The *experience-based human-centered design lifecycle* is an approach proposed by Metzker and Offergeld [MO05]. They identified that a main problem of several user-centered design processes is their premise that they completely replace established development processes of a company in order to improve the company's orientation towards user-centered design. In contrast, the experience-based human-centered design lifecycle integrates with existing development processes. It consists of a reference model, which lists recommended user-centered design activities, and an *introduction-establishment-improvement (IEI)* process model that describes how to select appropriate user-centered design activities and apply them in the existing development process. The IEI process model is iterative, so that the selection and application of user-centered design activities can be continuously improved.

Pyla et al. present a system called *Ripple*, which allows usability engineers and software engineers to follow their own lifecycles in a concurrent fashion [PPAH05]. The system keeps track of the current state of development of both groups and issues synchronization messages whenever collaboration is necessary. This is the case, for example, when both groups need to schedule meetings with users or when a document of one group has changed and needs approval from the other group. Lifecycle descriptions and development artifacts need to be stored in the Ripple system so that it can keep track of the current status of development and automatically generate synchronization messages.

2.4.2 Approaches with Focus on Artifacts

Adams et al. describe the *Usability & Software Architecture (U&SA)* that takes usability issues into account during architecture design [ABJ05]. They argue that usability decisions often not only influence the visual appearance of an user interface, but also require support in the architecture of a software system. If usability is not taken into account during architecture design, changes required to improve usability later on can be difficult to realize without changing the underlying architecture. They developed a set of *architecturally-sensitive usability scenarios*, which are actually on the level of abstraction of use cases, and which describe user-system interactions that require support from the system architecture. Each use case is annotated with information about its benefits to the user, like being able to undo interactions, and the functionality the system must provide in order to support the use case, like logging all user interactions.

Based on this information, developers can decide whether the benefit to realize a use case is worth the effort for the current project, and which steps are necessary in order to realize it. Adams et al. applied their approach in the context of an industrial project at NASA, where they developed a multi-user system to support scientists and engineers on the Mars Exploration Rover mission.

The potential for collaboration between HCI and SE professionals based on use cases has also been examined by Kujala [Kuj05]. In her industrial studies, she found that use cases were often written by software engineers who had no user contact. This resulted in use cases that were very technical and did not match the users' contexts, which made them hard to understand for users. She describes a three step approach in order to integrate user-centered design with use case driven software engineering. In the first step, small-scale field studies consisting of interviews and observations are conducted in order to get to know the users and their working context. Second, *user need tables* are created from the findings of the field studies. These list problems that are known or openly expressed by users with current processes as well as possibilities for improvement, which are typically more implicit. Last, use cases are derived based on the user need tables that have been developed.

The Usability & Software Architecture and the approach described by Kujala utilize use cases written in natural language for describing interactions between user and system and are thus understandable for all stakeholders. Due to their level of abstraction and the inclusion of alternative flows of interactions, however, use cases can lead to different interpretations and misunderstandings. To compensate for these problems, this dissertation relies on the use of scenarios, which focus only on a single, concrete flow of interactions.

Carter et al. [CLSF05] try to bring together usability engineers and software engineers by merging the artifacts that are created using the *Putting-Usability-First (PUF)* methodology [Car97] with the Unified Modeling Language (UML). PUF has the concepts of *users, scenarios, tasks, content* and *tools*. These are mapped to UML as follows: users are mapped to actors, scenarios and tasks are both mapped to use cases, and content and tools are mapped to attributes and methods, respectively. As the concepts in the PUF methodology contain more information than what can be expressed using standard UML elements, stereotypes are created and applied to the respective UML elements so that the additional information from PUF can be stored. This kind of mapping ensures that information that has been gathered using the PUF methodology gets forwarded into the development phase of a project and is not lost during a transition from PUF to UML.

The *User Engineering* process of IBM also utilizes UML in order to improve collaboration between usability engineers and software engineers [Rob05]. UML is used from the beginning of a project in order to capture usability related information. The use of UML as a notation familiar to software engineers is meant to foster understanding between usability engineers and software engineers. The User Engineering process consists of the phases *Business Opportunity, Understanding Users, Initial Design, Development,*

Deployment and *Life Cycle*, where *Life Cycle* refers to capturing user feedback to the deployed system. The author notes that these phases do not describe a waterfall-like process, but that it is possible to return to a previous phase at any time if omissions or errors are identified.

UMLi provides an extension of UML 1.1 that allows to model interactive applications [SP00, dSP03]. The extension is focused on the development of form-based user interfaces and aims at user interface developers who are used to working with modeling languages. UMLi introduces new elements to UML in order to be able to create abstract user interface definitions and adds a *user interface diagram* that visualizes the hierarchical relationship between user interface elements. User interface elements are identified from use case descriptions by inspecting the verbose description for verb forms denoting an interaction between user and system, like “user enters” or “system displays”. UMLi also extends the meta-model of activity diagrams with simplified notations to express order independent selection of actions, optional action selection and repeated execution of actions, which are identified as being common interactions in interactive applications. UMLi further strengthens the importance of activity diagrams by using them to interconnect use cases, where activities describe the flow of events of the use cases, the related user interface elements and domain elements.

The *Capability Maturity Model (CMM)* [Hum89] describes how mature a software organization is regarding their software development processes. CMM defines five levels of maturity, starting at level 1, the *initial level*. At this level, no management techniques are in place, appropriate software engineering techniques are only used inconsistently, and most development follows a trial-and-error approach and relies on the experience of the developers. Projects are often over time and budget, and project success is generally unpredictable. Organizations in level 2, the *repeatable level*, have a process in place that allows to repeat successes from previous projects if the project settings are similar. In level 3, the *defined level*, a process has been established that is standardized and documented, and which gets tailored to the needs of each specific project. In level 4, the *managed level*, measures about development process and product quality are collected and controlled, and in level 5, the *optimizing level*, processes are continuously improved based on these measures.

Wisdom (Whitewater Interactive System Development with Object Models) [Nun01] is a development method that provides a process, a model architecture and a notation. It aims at small software development companies, i.e. companies with less than 50 employees, who are currently operating on CMM level 1, following a custom-made process or no process at all, and that are in need of a process that is able to cope with constant change. The process part of Wisdom defines an evolutionary prototyping development model with a focus on user-centered design. Wisdom tries to bring small software companies to CMM level 3, where they take the Wisdom process model and adjust it to the needs of their current project. The notational part of Wisdom is realized as an extension of UML so that developers with knowledge of UML can build upon their knowledge. Wisdom, however, restricts itself to a subset of UML in order to reduce the

inherent complexity of the full UML specification, and adds missing elements that are needed to express requirements for the design of the user interface. The model architecture of Wisdom defines and relates the various different models it uses in order to support a user-centered design approach.

Blankenhorn and Jeckle describe a UML 2.0 profile that allows to model the static aspects of a graphical user interface [BJ04]. In contrast to approaches that only allow to model the hierarchical organization of the graphical user interface, e.g. a window that contains a form that contains text fields, their approach also captures the concrete arrangement of elements in two-dimensional space, e.g. the position and size of elements on the user interface. As their profile conforms to the UML Diagram Interchange specification [Gro06], the resulting models can be serialized and exchanged between project participants.

All of the afore-mentioned approaches build on UML as a basis to bring HCI and SE together. While this is suitable for communication between usability engineers and software engineers, users may not be familiar with the semi-formal notations of UML. The Wisdom approach additionally encourages the use of prototypes for evaluation, however, it defines no clear relation between prototypes and other requirements specification artifacts like scenarios, as the SCRIPT framework does.

Chapter 3

Requirements

Requirements stand at the beginning of every development project, no matter whether for a physical product, a software system or a combination of both. Requirements describe the desired features and properties that the resulting artifact should possess. Functional requirements describe the functionality that should be provided, while non-functional, or quality requirements describe properties that the resulting product must have.

As described in Section 2.2, the activity that is concerned with gathering all relevant requirements for a development project is called “requirements elicitation”. This activity is crucial to the whole development project, as subsequent activities build upon the requirements specification that is produced during requirement elicitation. Errors that are introduced and stay unrecognized during this phase can propagate throughout all subsequent activities until they are detected by the client or users once they start to work with the deployed system, which can then lead to costly changes if large parts of the system need to be reworked. [Boe81]

Requirement elicitation is a challenging task, as it involves both stakeholders, who have knowledge in their application domain, but do not know much about the technical details of development, and developers, who have technical knowledge in building a solution, but often only little experience in the application domain. The differences in knowledge and vocabularies that these two groups of people use can lead to misunderstandings, which can result in erroneous requirements specifications. Nevertheless, involving stakeholders in the requirements elicitation process is essential [HB95]. Developers often prefer formal notations for requirements, as they are easier to test for consistency, completeness, correctness and unambiguity. Many stakeholders, however, are not familiar with formal languages, so that alternative representations of requirements need to be used in order to involve them into the requirements process.

This chapter is organized as follows: Section 3.1 presents approaches that make formal requirements understandable for stakeholders without knowledge in formal languages via visualization and simulation. Section 3.2 focuses on use cases and how they can benefit from being enhanced with visual representations. Section 3.3 explains the concepts behind scenarios, in what contexts they can be applied, techniques for deriving formal

specifications from natural language representations of scenarios, and augmentation or representation of scenario using visual means.

3.1 Requirements Visualization and Simulation

The main goal of requirements visualization is to evaluate formalized requirements in order to detect errors before development starts. As stakeholders are typically not familiar with formal notations for requirements, visualizations can help them in understanding and evaluating requirements models that have been constructed by developers.

Pérez and Valderas describe an approach for eliciting requirements for pervasive systems [PV09]. It allows users to specify the requirements of the system themselves using a visual editor, based on an underlying model of available services and devices. The editor shows a visual representation of the target environment, e.g. the map of a house, and the user selects which services they want in which room, e.g. automatic illumination. The editor then determines the devices that are necessary for providing the selected services, e.g. a motion detector, and asks the user to place the required devices on the map. Users can receive guidance by a requirements engineer if necessary. The immediate visual feedback via the editor actively involves users in the requirements process and enables them to evaluate the resulting requirements specification to correctly express their needs.

Pseudo software is another approach at making requirements accessible for stakeholders [JC07, JC10]. It provides a unified requirements repository where users themselves enter requirements about presentation, navigation between screens, input constraints, business logic and test cases. To store this information, they use *pseudo software* which only provides the user interface, but with no logic implemented. Users input the requirements by adding it to the respective elements of the user interface, e.g. defining the range of legal values for a text field. Requirements are not stored in a formal model, however, but as plain text descriptions.

Requirements simulation is an extension to requirements visualization by taking timing into account. The SCR (Software Cost Reduction) method [HKLB98, BJHW00] provides tools for capturing software requirements for safety-critical systems. In order to evaluate the correctness of the requirements, simulations can be executed. SCR allows to attach domain specific front-ends, such as the image of the cockpit of a jet, so that domain experts can operate and evaluate the simulation in a setting that they are familiar with. Another example is given by Van et al. [VvLMP04], who present an approach that is based on goal models and state machines to simulate requirements. Their approach allows users to interact with the simulation and evaluate different flows of events. UML state diagrams as well as real-world images can be attached to the simulation, e.g. the images of a train with doors open (state 1) and doors closed (state 2) in the context of specifying requirements of a train control system.

All of these approaches are similar to the SCRIPT framework in that they aim at making formal requirements understandable for stakeholders without knowledge in formal notations. None of them, however, utilizes scenarios to gather requirements, which is one of the main focuses of this dissertation.

3.2 Use Case Visualization

Scenarios and use cases stand in strong relation to each other, although their focus is slightly different. Both are written from the point of view of the user, using only terms of the application domain and without reference to implementation specifics. A scenario, however, describes one single, concrete flow of interactions between user and system that possibly spans multiple system functionalities, while a use case describes the interactions regarding a single functionality of a system, but with all possible variations.

Use cases can be written with different levels of granularity regarding the details of interactions. Essential use cases [Con95] focus on interactions between user and system on an abstract level without specifying any details about how the user interacts with the system. Essential use cases focus solely on the goals of the user, e.g. identification, and not on any details of the realization of the system, e.g. identification using a keycard or via an iris scan. Essential use cases, like use cases in general, are written solely in the language of the user and the application domain, which allows users to relate to the interactions described in the use case. Utilizing essential use cases prevents to narrow down on a single solution too early in the beginning of development by keeping the description of interactions on an abstract level. This advantage is a problem at the same time, as the abstractness of essential use cases can lead to diverging interpretations by clients, users and developers, and thus result in misunderstandings.

Use case descriptions can benefit from augmentation with appropriate visualizations [GFHR95]. Especially when it comes to define the detailed layout of the user interface, it is much more efficient if this is not done in textual formal, but rather with appropriate images that display the user interface [KS07].

One example where this has been applied is the storyboard process for the selection of commercial-off-the-shelf (COTS) components presented by Gregor et al. [GHO02]. They employ use cases to help users determine the required functionality and evaluate how much of that functionality can be provided by using existing software solutions. Use case visualizations are created by making screen shots of the existing software solutions and enhancing them via graphic manipulation, or by creating mockups of completely new user interfaces. These images are then arranged on a poster size graphic that shows all steps of a use case from the user interface perspective.

The *Use Case Workbench (UC Workbench)* is a tool presented by Nawrocki and Olek [NO05]. It allows to couple steps in a use case with images and generate a prototype that enables users to step through the flow of interactions of the use case. Whenever

the user reaches a use case step that has been coupled with an image, it is shown next to the textual description of the use case.

A more sophisticated approach is the *Fast Feedback* technique presented by Schneider [Sch07]. It aims at capturing and validating use cases in requirements elicitation meetings with stakeholders. It utilizes a tablet computer that is used during the meeting for noting down information about use cases and for sketching user interface drafts alongside. The technique is backed by a system that allows to immediately connect use case steps and user interface sketches, and that provides a step-by-step animation of the flow of events on the user interface, similar to *UC Workbench*. The interrelation between use case steps and user interface drafts, however, is more elaborate than with *UC Workbench*. It is not only possible to connect whole user interface sketches with use case steps, but also regions on the user interface sketch, which are then highlighted by a red rectangle. Stakeholders can pretend to interact with the sketches by drawing on them. As no application logic can be defined with the *Fast Feedback* technique, the system does not react to the user input, but instead the stakeholders' interactions are recorded for later analysis.

Development of use cases and user interface designs can also happen in parallel, as described in the *user interface based design process* presented by Mrdalj and Jovanovic [MJ02]. They rely on the availability of use cases and related early user interface visualizations in order to derive various analysis artifacts such as UML class diagrams, sequence diagrams, collaboration diagrams and state chart diagrams from them.

The approaches presented in this section are based on the application of use cases, which are similar to scenarios in that they describe interactions between user and system. Use cases, however, do not focus on a single, concrete flow of interactions, but rather describe interactions on a more abstract level and include alternative flows of interactions, thus bearing the risk of misunderstandings between developers and stakeholders due to differing interpretations. Hence, this dissertation utilizes scenarios instead for describing concrete interactions between user and system to ensure that all stakeholders have the same understanding about how interactions between user and system are going to take place.

3.3 Scenarios

According to Carroll, a scenario is “a narrative description of what people do and experience as they try to make use of computer systems and applications” [Car95]. A scenario does not focus on describing only a single functionality of a system, but rather describes a concrete flow of interactions between user and system that is understandable by clients and users. In order to keep the complexity low, alternative flows of events are not considered within a single scenario, but instead multiple separate scenarios must be used to describe them.

Scenarios can be used with different purposes, from requirements elicitation to strategic planning, and cover various timespans, from a whole year over a day down to single keystrokes [GC04]. Scenarios can be used to describe the current situation in form of *as-is* scenarios, or describe how future interactions are supposed to take place in the form of *visionary* scenarios [CR92]. In visionary scenarios, the flow of interactions describes the requirements on a system, i.e. what functionality a system must provide in order to make the interactions possible. Scenarios have already found acceptance for use in software development [HCR05], but it has also been recognized that working with scenarios does not alleviate the need for continuous evaluation and adjustment of requirements during the lifetime of a project [CRCK98]. Apart from their use for communication, scenarios can also serve as input for other artifacts, such as prototypes, as has been reported by Hertzum [Her03], although they did not elaborate on the relation between scenarios and prototypes.

Scenarios are often in use with slightly different understandings of what is actually meant by the term “scenario”. Rolland et al. identified 12 different scenario-based approaches and evaluated them according to a framework with 4 dimensions, which are *form*, *contents*, *purpose* and *lifecycle*. *Form* describes the representation of scenarios, *contents* describes which information is expressed with scenarios, *purpose* describes the reason for which scenarios are used, and *lifecycle* describes how scenarios are captured and modified. They found that many approaches use natural language in order to describe concrete situations or behaviors in an open-ended and informal way. [RBAC⁺98]

Using narrative scenarios has the major advantage that it allows to involve all stakeholders without requiring prior training on some formal language. For subsequent development steps, however, a formalized scenario representation is desirable, so that developers can directly access relevant parts of a scenario without requiring them to read large amounts of plain text when they are looking for an information about an interaction that is described in a scenario. Additionally, a formalized scenario representation also allows to relate it to other requirements or system models on a fine level of granularity, thus improving requirements traceability. Section 3.3.1 presents approaches for representing scenarios in a formalized way.

Additionally, natural language used in narrative scenarios is sometimes not expressive enough to describe complex situations or revolutionary ways of interacting with a system. In these cases, scenario visualizations can support building a shared understanding among all stakeholders, similar to the application of use case visualizations that have been presented in Section 3.2. An overview of existing approaches for scenario visualization is given in Section 3.3.2.

3.3.1 Scenario Formalization

Scenarios are not only useful during requirements elicitation, but also for subsequent steps of software development. In these cases, developers sometimes prefer formalized representations over textual representations, as formalized scenarios can be transformed

into other models like state-based models for further processing [LDD06]. An overview of existing scenario formalizations is given in this section.

The Unified Modeling Language (UML) 2.0 provides three types of diagrams for modeling interactions: *sequence diagrams*, *communication diagrams* and *interaction overview diagrams* [Gro12b]. A notation similar to UML sequence diagrams are *Life Sequence Charts* [DH01] that allow to model more sophisticated situations, including conditions that need to hold either once or during the whole execution.

ScenarioML is a notation described by Alspaugh, which allows to formally represent the flow of interactions a scenario consists of [Als05]. It is based on Allen's *interval algebra* [All83] that allows to define how time intervals relate to each other, like occurring sequentially or ending at the same time, and it uses XML syntax. *ScenarioML* allows to explicitly state temporal relations between interactions in a scenario that otherwise might get overlooked.

Elkoutbi et al. present an approach that is based on UML 1.0 [EKK99, EK00, EKK06]. For requirements acquisition, they use collaboration diagrams to specify scenarios and use cases, and class diagrams for application domain modeling. They describe an algorithm to automatically generate user interfaces and prototypes for evaluation based on the created models. Their approach, however, requires a large scale modeling effort upfront.

Hsia et al. describe an approach that uses trees of scenarios, where each node stands for a single interaction and a path from the root to a leaf represents a single scenario. These scenario trees are then converted into regular grammars. Their goal is to arrive at a "precise, unambiguous, consistent and [...] complete" set of scenarios. These scenarios are also used for requirements validation via generation of prototypes, while they do not provide information about how this is to be achieved in detail. [HSG⁺94]

All of these approaches focus only on formal representations of scenarios, which has the drawback that stakeholders can no longer work with them directly without prior training in the formal language used. The SCRIPT framework compensates for this problem by using a model-based notation of scenarios that also provides a narrative representation.

An approach that is based on natural language processing is presented by Kof [Kof07, Kof08]. The goal is to analyze narrative scenarios and automatically identify missing objects. Additionally, a message sequence chart (MSC) [GGR93] can be automatically generated that depicts the flow of interactions that are described in the underlying scenario.

Kaindl and Jezek developed an approach that analyzes scenarios in order to determine which kind of user interface elements, like buttons and text fields, should be used in the user interface [KJ02]. They formalize scenarios by analyzing each step in the flow of interactions of a scenario and relate it to one or more functional requirements that the system needs to fulfill. Functional requirements are then decomposed into low-level interactions and classified according to their type of interaction, e.g. data input or output. Based on this classification, classes of suitable user interface elements are derived, e.g. text fields for data input.

While the last two approaches are based on the use of narrative representations of scenarios, they do not provide interactivity so that users can evaluate the interactions themselves.

3.3.2 Scenario Visualization

Scenarios are often represented as textual narratives using natural language. Text, however, is sometimes not expressive enough to unambiguously describe complex situations. As a Chinese proverb says “A picture is worth a thousand words”, it is often useful to augment textual scenario descriptions with supporting images. In other cases, where scenarios are captured using a formal language, visualizations of the user interface or the interactions described in a scenario can help in providing clients and users with a representation they can understand without prior training, as has been described in Section 3.1 for requirements in general.

The SCRIPT framework uses prototypes as a form of visualization for the scenario models that are created from its underlying scenario meta model. While prototypes are discussed in more depth in Chapter 4, this section gives an insight into existing scenario visualization techniques.

A simple technique for scenario visualization is to attach images like static design drafts of the user interface to the scenario description, either only for selected interactions or all interactions of the scenario.

Alspaugh et al. evaluated the use of social agents to visualize scenarios in a virtual 3D environment [ATB06]. Their approach is based on scenario descriptions in the formal language *ScenarioML*, which has been described in the previous section. Actors and entities of scenarios are mapped to virtual characters, and animations are automatically generated that visualize the interactions that are described in the scenario. Additionally, a text-to-speech system reads the textual descriptions of interactions in parallel to the animation. Although an initial evaluation could not show that the identification of errors with this approach is higher compared to text-only scenario descriptions, the inspection of the animations during preparation revealed problems like missing interactions that might have stayed undetected otherwise.

An approach that uses real-world video recordings of scenario interactions called *Software Cinema* is described by Creighton [Cre05, COB06]. Its focus is not on interactions between user and system on the level of the user interface, as it is of interest in this dissertation, but rather on physical interactions between users and system. Hence, it is tailored towards systems that do not follow traditional desktop interaction patterns, but rather support mobile users that are in need of different interaction paradigms. Formalisms underlying the approach allow to use the resulting videos for subsequent development activities.

Another scenario visualization is part of the *scenario-based requirements analysis method (SCRAM)* [Sut03]. It uses so called *concept demonstrators* that can playback a predefined script, which represents the flow of interactions of a scenario. SCRAM allows

to use images in order to visualize the interactions of the scenario. The resulting concept demonstrator is not intended for user operation but merely for demonstration purposes in order to stir discussion with users.

All of these approaches enrich scenarios with visualizations in order to improve the understandability of scenarios and ensure a common ground for discussion with stakeholders. None of them, however, provides interactivity so that stakeholders can actively experience the interactions that are described in a scenario, as it is possible with the SCRIPT framework.

Chapter 4

Prototyping

Taken literally, a prototype is the “first of a type”, a notion that originated in the context of the creation of physical goods. In that context, a prototype is an artifact that shares all key features and attributes of the final product and serves as a template for mass production [Flo86]. In the context of software development, the major challenge is not the reproduction of a finished system, but rather the development of the “first” instance. Here, prototyping is used to evaluate parts of a system before the complete system has been built in order to gather insights on various aspects of the system.

Rapid prototyping emerged as a term for porting the activity of creating (physical) prototypes into the domain of software engineering, as explained by Tripp and Bichelmeyer [TB90]. According to them, rapid prototyping is different from “classical” i.e. physical prototyping in that the reason for prototyping is not to mitigate the consequences of error (like an airplane prototype, which is built in order to ensure the airplane is airworthy), but to increase efficiency of the development effort. Additionally, rapid prototyping aims at the discovery of additional development goals and not only at the satisfaction of goals that are already known.

Already in 1984, Alavi presents findings from a study [Ala84], which showed that prototyping is beneficial as it provides a ground for discussion between stakeholders. As one of the interviewed project manager stated: "The users are extremely capable of criticizing an existing system but not too good at articulating or anticipating their needs." This corresponds to observations made by Boehm, who describes this as “I’ll know it when I see it” (IKIWISI) [Boe00]. It describes users who can only really express their needs once they start to evaluate a prototype, and not by defining them upfront. Prototypes help users to make decisions, involve them into development and enable them to evaluate requirements specifications more effectively than with paper-based documents [AH93]. Andriole also emphasizes the importance of prototyping and proposes a development process that starts with initial requirements and system design and which then utilizes prototypes for refining both [And94]. The importance of prototyping has also been recognized by Wasserman [Was96], who names it one of the eight fundamental ideas of software development. He sees its advantages not only in the area of the development of “interactive information systems”, but also for developing appliances like copy machines, automated teller machines and the like, as they all have user interfaces.

Prototyping has become an integral part of lifecycle models like Boehm's *spiral model* [Boe88] or the *Usability Engineering Lifecycle* of Mayhew [May99]. The International Organization for Standardization (ISO) also recognized the importance of prototyping and recommends to use prototypes to gather feedback from users regarding design alternatives already during early phases of development, as described in ISO 9241-210: "Human-centred design for interactive systems" [ISO10].

Prototyping, however, can only be an effective means if the goals that are to be achieved by using prototypes have been identified upfront. Different kinds of prototypes are suitable for evaluating different aspects of a system. In order to support the decision for the kind of prototype most suitable, various people proposed different categorizations for prototyping. A general distinction can be made between categorizations that analyze prototypes as artifacts, and categorizations that refer to the process of prototyping [BBLZ96].

The remainder of this chapter is organized as follows: Section 4.1 gives a short discussion about prototyping and design, which is a relevant background to better understand the motivations that stand behind the development of the SCRIPT framework. In Section 4.2, approaches for classifying prototypes as artifacts are presented, followed by classifications that refer to prototyping as a process in Section 4.3. Section 4.4 presents various techniques for prototyping, and in Section 4.5, existing prototyping tools are presented. Section 4.6 is dedicated to approaches for making knowledge available that is contained in prototypes.

4.1 Prototyping vs. Design

Among other uses, prototyping can be utilized to support the development of the user interface. When it comes to define those parts of an artifact or system that are visible to and accessible by users, this activity belongs to the domain of design, and as such some general design principles are also applicable for the design of interactive systems. According to Buxton, one important aspect in design is to actively explore and broaden the space of possible solutions instead of narrowing on one concrete solution too early during development [Bux07, p.389]:

"The role of design is to get the right design.
The role of usability engineering is to get the design right."

With *usability engineering*, he refers mainly to prototyping as a usability engineering technique, especially in the form of iterative prototyping, where feedback gathered with a prototype of version n feeds into the development of a refined prototype of version $n+1$. This iteration on a specific artifact serves to "get the design right". In contrast, design strives to explore multiple alternative, potentially fundamentally different solutions to provoke discussion and stir new ideas and insights, which is the goal of "get[ing] the right design".

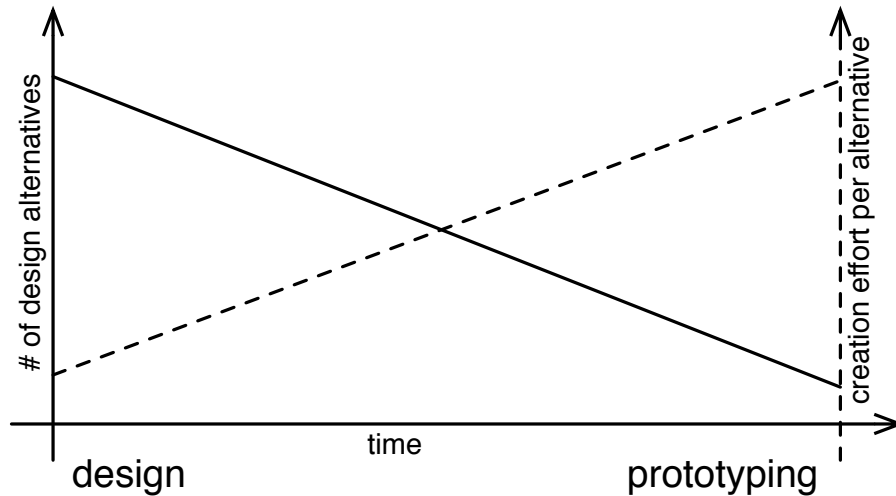


Figure 4.1: Relation between *number of design alternatives* and *creation effort per alternative*

The relation between design and prototyping is visualized in Figure 4.1. At the beginning of a project, the number of design alternatives is supposed to be high, with only little effort needed for the creation of each design, so that many alternatives can be created and disposed easily. Through testing and evaluation, more appropriate designs are distinguished from less suitable ones, so that the number of design alternatives slowly reduces. The selected designs, however, get refined in more detail, thus raising the creation effort for the remaining design alternatives.

The importance of evaluating multiple design alternatives has also been examined by a study conducted by Tohidi et al. [TBBS06]. They observed that the ratings users gave on an absolute scale to user interface designs were different, depending on whether they were presented with multiple alternatives or only a single design. Most importantly, the design that has been rated worst by the group with multiple design alternatives, received much better scores from the group that was only presented that specific design, which might have led to false conclusions if the study had been the basis for the development of a real product.

For the quick and easy creation of design alternatives, Buxton promotes the use of sketches [Bux07]. It is important not to confuse sketching with prototyping, although sketched designs of a user interface may well become part of a prototype. Sketching, however, pursues different targets than prototyping.

First, sketching is a mental activity that helps designers to reflect on their current knowledge, contained in their mind. While the creation of a sketch is driven by the current knowledge of a designer, the representation of knowledge in form of a sketch leads to new insights that result from “reading” the sketch, and thus leads to a modification of the knowledge in the designer’s mind. The interplay between mind and sketch is

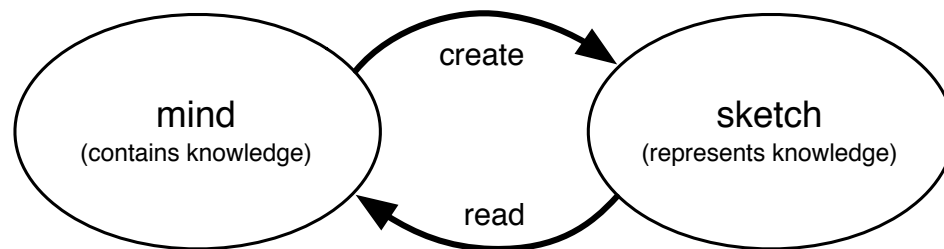


Figure 4.2: Interplay between *mind* and *sketch*. Adapted from [Bux07]

visualized by Figure 4.2. This use of sketching has also been observed by Tversky [TSLD02].

Second, sketching is a design activity and hence takes part in the very beginning of a project, where the goal is to create as many different alternative solutions as possible. As such, a sketch differs from a prototype in various aspects, e.g. in that it tries to explore rather than to refine, or that it is tentative rather than specific. The aspects in which sketches differ from prototypes can be seen in Figure 4.3. Note that the depicted terms denote the extremes of a continuous spectrum between sketches and prototypes.

The distinction between sketching and prototyping becomes blurred when it comes to paper prototyping, which relies heavily on the use of user interfaces that are “sketched” on paper. Buxton sees the major difference in how the sketched images are used. If they are used by designers themselves to evaluate their ideas, or with a user in a setting that allows to make instant modifications once new ideas come up, then Buxton qualifies this as sketching. Once a paper prototype is evaluated in a more formal setting with multiple users that are all presented the same user interface and where the focus is on evaluating that specific design, that is, find out if the “design is right”, then this qualifies as prototyping. [Bux07, p.381]

The SCRIPT framework takes a role similar to paper prototyping and is located between design and sketching on the one hand, and usability engineering and prototyping on the other hand. It supports the creation of multiple alternative design ideas with as little effort as possible. This is emphasized by a strong orientation on classical paper prototyping and the goal of enabling designers to apply it without requiring major prior training. Additionally, the resulting artifacts are suitable for evaluation with users and further refinement in subsequent development activities.

4.2 Prototypes as Artifacts

This section gives an overview of approaches that analyze prototypes as artifacts and classify them based on attributes that are inherent to them.

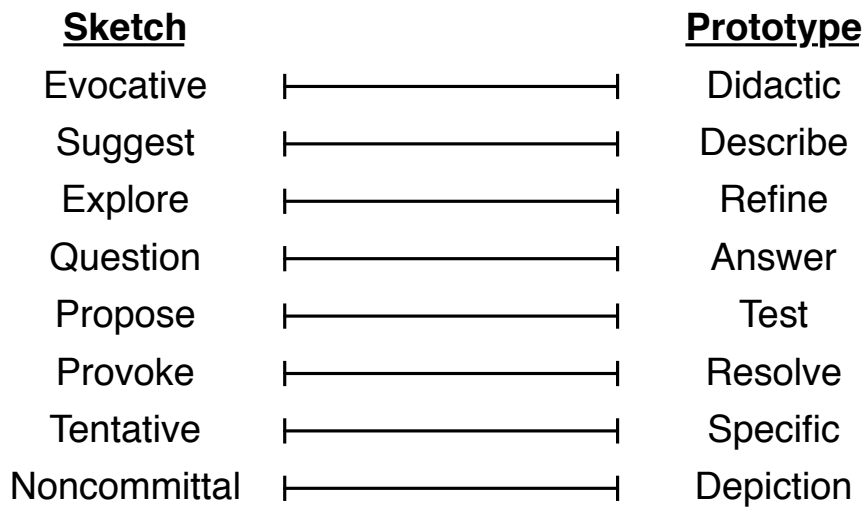


Figure 4.3: Continuum between Sketch and Prototype. Adapted from [Bux07]

4.2.1 Categories of prototypes

Lichter et al. provide a way of categorizing prototypes according to their experiences from the use of prototyping in the industry [LSZ94]. They propose a categorization into *presentation prototypes*, *prototype propers*, *breadboard prototypes* and *pilot systems*.

Presentation prototypes are used for acquisition before a project has started and serve to convince clients that a system can be built, either regarding the use of some technology or that the user interface is able to fulfill user requirements. As they are mainly used for persuasion, they are typically created with a “quick-and-dirty” approach and are disposed of after they have served their purpose.

A *prototype proper* is used during requirements engineering and can help to clarify questions that arise during this activity. It realizes parts of the user interface or selected functionalities where uncertainties exist and helps to resolve them in order to arrive at a sound requirements specification. As such, it is used for communication between all stakeholders, including clients and users.

A *breadboard prototype* focuses on the evaluation of technical details of the realization of a system. Prototypes of this kind are often used implicitly without being named as such, and they are mostly used for discussions between developers without the inclusion of clients or users. Technical solutions found via a breadboard prototype may transition into the final system implementation.

When a prototype is mature enough to be used in its destined context, it is called a *pilot system*. While it may still only provide a subset of the complete set of features defined, the features already available need to be realized to their full extend, which includes satisfying requirements on performance, usability and the like. The close rela-

tion between a pilot system and the final application blurs the strict distinction between prototype and application.

4.2.2 Horizontal and Vertical Prototypes

In system designs that follow a layered architectural style, prototypes can be classified into *horizontal prototypes* and *vertical prototypes* [LSZ94]. A layered architectural style defines a system in terms of layers that group similar functionality and where each layer must only depend on functionality provided by the same layer or underlying layers, possibly even constrained to the layer directly beneath it. A common form is the three-layered architectural style that consists of a database layer at the bottom, an application logic layer, and a user interface layer on top.

A *horizontal prototype* is a prototype that is constrained to one layer of the system, but where the selected layer is realized completely. While often employed as user interface prototypes, a horizontal prototype can as well prototype any other layer of a system. If a horizontal prototype is created in the former sense, it realizes the complete user interface, but without any functionality behind it.

A *vertical prototype* in contrast is not confined to a single layer, but to a subset of the functionality specified in the requirements specification. For this subset, however, the implementation covers all layers from top to bottom to prove for technical feasibility.

4.2.3 Prototype Fidelity

Prototypes are often classified regarding their fidelity. The fidelity of a prototype is determined by the attributes of a prototype that are visible to a user, not by the technical details that relate to the creation of the prototype [RSI96]. The distinction between *low fidelity* and *high fidelity* is a rather vague, nevertheless wide-spread approach. A more fine-grained classification can be achieved by evaluating multiple dimensions independently.

Low Fidelity and High Fidelity

Rudd et al. [RSI96] propose a classification of prototypes, which is mainly focused on prototypes that have user visible components and that are intended for discussion with clients and users.

According to their classification, a *low fidelity prototype* is a prototype that can be produced with very little effort and which provides limited or no interaction at all. It is used for exploring screen layouts and design alternatives, including details such as colors and control placement, rather than user interaction. As such, it is supposed to be operated by a facilitator, who either follows through a predefined scenario for demonstration purposes, or who executes requests from users. Interaction is mostly constrained to exchanging sheets of paper of a paper prototype or advancing slides of a

presentation to simulate screen flows. The main advantage of low fidelity prototypes are that they can be created fast and cheap and thus allow to explore various alternatives instead of narrowing down on one solution too early. On the other hand, they are too coarse to guide developers in the implementation of the final system and can provide only very limited help should questions about realization details of the user interface arise.

In contrast, a *high fidelity prototype* looks and behaves like the final system with respect to the user interactions that it supports. Users can evaluate the prototype by themselves and interact with it as if the system already had been implemented. Quality requirements like performance, accuracy or security, however, might not be satisfied by the prototype. The focus of a high fidelity prototype is to give users the opportunity to evaluate the flow of interactions and make suggestions for improvement of the user interface. As the user interface is already in a very mature state, developers can refer to it during implementation to clarify questions about the visual appearance and interactiveness of the user interface. High fidelity prototypes can also be used for marketing and training purposes or help in creating the documentation of the system. Their high sophistication comes at the price of massively increased costs for creation compared to low fidelity prototypes. Additionally, they tend to raise wrong expectations with the clients, who take the prototype for the final system and cannot understand why the actual development of the system might take quite some time after the prototype has been made available to them.

Multi-Dimensional Fidelity

The classification of prototypes into low and high fidelity bears the problem that it subsumes multiple orthogonal aspects of a prototype that deserve individual consideration. To overcome this one-dimensional “fidelity barrier”, McCurdy et al. [MCP⁺06] developed a classification of prototypes according to five dimensions, which are *level of visual refinement*, *breadth of functionality*, *depth of functionality*, *richness of functionality* and *richness of data model*.

Level of visual refinement describes how “finished looking” the user interface of a prototype is. It ranges from hand-drawn sketches to pixel perfect high resolution imagery.

Breadth of functionality describes how much functionality a prototype offers to its users. A prototype that ranges on the low end of this dimension provides only selected functionality, while a prototype on the high end offers a large amount if not all functionality that the final system is supposed to have.

Depth of functionality describes to how much detail a prototype has been realized concerning user–system interaction. A prototype that allows users to execute each and every step of the flow of interactions of a certain task is positioned on the high end on this dimension.

Richness of interactivity describes how precisely a prototype reflects the kinds of interactions the final system will support. Paper prototypes are located at the low end

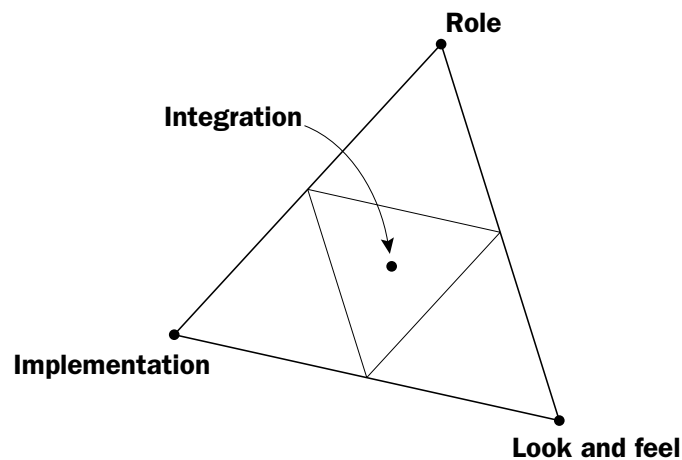


Figure 4.4: Aspects of prototyping by Houde and Hill. From [HH97]

of this dimension, while prototypes on the high end typically also involve increased implementation efforts.

Richness of data model describes how closely the data represented in a prototype resembles real data from the application domain, both in quantity and appearance. In cases where real datasets become rather large or complex, care must be taken not to oversimplify datasets used in the prototype. Otherwise, positive evaluation results achieved with the prototype may be non-representative for the appropriateness of the system with real data.

4.2.4 Focus of Prototypes

Houde and Hill propose a classification of prototypes that is not based on the details of its realization or appearance, but rather on the aspects of the system the prototype aims at [HH97]. They differentiate between *look and feel*, *role* and *implementation*, which are not mutually exclusive but span a triangle as can be seen in Figure 4.4, where a prototype may occupy any place in-between, depending on the degree to which it focuses on each aspect.

The *role* aspect describes how a system can support its users, that is, what role a system may take in the actions of its users or how it may enable previously impossible actions. The focus is not on the details of how the system actually looks and feels during the interactions, or how the interactions are realized technically, but rather on which interactions are possible at all and how they fit the users' contexts.

The *look and feel* aspect focuses on the concrete realization of a system with respect to the parts that are visible to a user. This includes both its appearance as well as the concrete types of interactions the system offers its users. Look and feel prototypes

are used to convey an idea about how the final system will look like and to enable users to evaluate different types of interactions. However, they rather focus on singular interactions and may not take into account the overall role of the system in the users' context.

The *implementation* aspect describes how much a prototype focuses on the technical feasibility of a system. Prototypes that focus solely on this aspect usually are used only internal to the development team, however when it comes to evaluate quality requirements like performance, user involvement can be useful also for an implementation prototype.

Integration prototypes do not constitute a prototyping aspect by themselves, but rather a combination of role, look and feel and implementation. Prototypes of this kind strive to evaluate all aforementioned aspects in more or less equal depth. This results in increased efforts for building an integration prototype, so that this kind of prototype is used rather rarely and typically only during later phases of development.

4.3 Prototyping as Process

In contrast to analyzing prototypes as artifacts, prototyping can be analyzed from a process perspective, where the focus is on the goals that are supposed to be achieved by employing prototyping. The following categorization of prototyping processes is based on findings from Floyd [Flo86] and Graham [Gra94]. This kind of classification is complementary to the classification of prototypes as artifacts.

4.3.1 Revolutionary prototyping

The goal of revolutionary prototyping, which is sometimes also called “exploratory” or “throw-away” prototyping, is to resolve communication problems that might come up during requirements specification, either due to a lack of domain knowledge of the developers, or originating from the fact that users might not be able to imagine in which ways the system can support them.

Prototypes that are used in this context are supposed to be created fast and without much development effort. The development environment for revolutionary prototyping can be substantially different from the target environment, where the final system is to be employed. This may be due to the fact that the target environment enforces the use of programming languages that are not suitable for quick and easy prototype generation, or because the target environment has not been yet decided at all.

Prototypes should allow users to perform a typical task of their every day work practice with help of the prototype. Instead of only presenting one solution, ideally multiple alternatives should be presented so that the most accepted features of each prototype can then be integrated into the final system. However, it should be clearly communicated to clients and users that these revolutionary prototypes only serve to stir ideas and input

for requirements specification, and might not become part of the final system, neither their underlying code nor the functionalities they represent.

4.3.2 Experimental Prototyping

Experimental prototyping aims at clarifying open issues about the system under development. These issues may relate to the interaction between user and system or the technical realization of the system. Prototypes used for experimental prototyping come in different forms. They range from prototypes that realize the complete user interface without any underlying functionality in order to get feedback from the user, over prototypes that only support a single task but with full functional support, to prototypes that provide the complete set of functionality as the final system, but quickly written in an intermediate language that does not fulfill performance requirements, or ignoring security constraints. In contrast to prototypes from exploratory prototyping, prototypes used for experimental prototyping can become part of the final system if this is technically possible and if the code fulfills quality standards.

4.3.3 Evolutionary Prototyping

Evolutionary prototyping is based on the experience that requirements for a system are in constant change, as the context surrounding a system changes constantly, which thus leads to the arising of new requirements. The productive usage of a system itself also stirs the identification of new or altered requirements. Therefore, evolutionary prototyping describes an approach where a system is built up gradually instead of being completely specified upfront and implemented in one huge effort. By building and deploying a system step-by-step, each iteration of the system can serve as a prototype to gather feedback for the next iteration, thus ensuring maximum alignment to the users' needs. However, this also requires users to accept the possibility of changes to existing functionality and developers to follow a rigid work style to make evolutionary prototyping technically feasible. In contrast to revolutionary prototyping, the development environment and target environment are similar if not identical with evolutionary prototyping.

4.4 Techniques for Prototyping

In this section, various techniques for prototyping are presented. While the previous two sections were concerned with rather abstract classifications of prototypes, this section presents concrete techniques for using prototypes during software development.

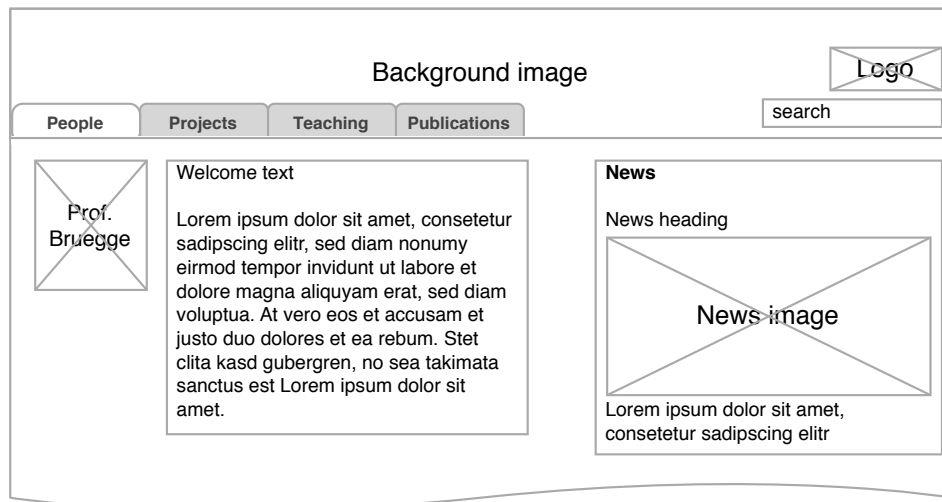


Figure 4.5: Wireframe of the web page of the Chair for Applied Software Engineering

4.4.1 Wireframe Prototyping

The term **wireframe** has its origin in the context of 3D modeling, where it denotes a visual representation of an object only by its supporting structure, with intermediate areas being empty and without its final color or texture applied. In the context of user interface design, it stands for a version of the user interface that is only concerned with general arrangement like size and positioning of user interface components, ignoring details such as typeface, font size or color. Images are often replaced by simple boxes with a cross in them and text is replaced by greeked text in order to focus attention to layout issues rather than design details. An example of such a wireframe is given in Figure 4.5. Wireframes can evolve from these rather abstract, low-fidelity (concerning visual appearance) graphics to high-fidelity, polished looking designs as design proceeds.

Wireframe prototyping describes the use of wireframes to lay a basis for the design of the user interface. A wireframe prototype can be restricted to a single wireframe of a screen of the system, a set of wireframes that supports following through a specific sequence of interactions, or a complete set of wireframes covering the whole user interface. Wireframe prototypes are easy to create as they can be drawn by hand or using any drawing software available. They provide no interactivity themselves but can be used for the creation of a digital prototype in case they are available in digital form. [AAB07]

4.4.2 Storyboard Prototyping

Storyboards are used in various disciplines and are most commonly known from the movie industry. In this context, they constitute a comic-like representation of the final movie and are used to anticipate costs and complexity during pre-production. [Wik12].

Apart from the movie industry, they have also made their way into business management to determine “brand touchpoints” that describe the situations in which users get in touch with a brand or company [SM10].

In software development, storyboard prototyping is used to depict a concrete flow of interactions between user and system, and has thus a very strong relation to scenarios. A storyboard prototype can start as a purely textual narrative and evolve by including images of the user interface [And89] or even video snippets of users interacting with the system. A storyboard prototype may focus on a small sequence of interactions belonging to one functionality, or it may be used to describe a larger sequence of related interactions that exemplifies how certain functionalities are required to work together. One advantage of using storyboard prototypes for software development is that they help all stakeholders in gaining a shared understanding of what the system is supposed to do, and to ensure that this focus does not get lost during development, even when requests for additional or altered functionality arise.

For cases in which storyboard prototypes are used to depict the context of system usage, Truong et al. conducted a study about best practices regarding certain aspects of the imagery used in a storyboard [THA06]. They analyzed the use of *accompanying text*, *depiction of persons*, *level of detail*, *number of panels* and the representation of *progression of time*. According to the best practices they derived, *accompanying text* should be used when presenting novel applications, *depiction of persons* should be used when feedback on interactions is desired and not on details of the user interface, the *level of detail* should be as abstract as possible in order not to confuse users with unnecessary details, the *number of panels* should be between 3 and 5, and *progression of time* should only be explicitly visualized if it is relevant for demonstrating a feature.

Unfortunately, the term “storyboard” is also used with a different, misleading interpretation. In these cases, it refers to a 2D canvas-like visualization that depicts a subset or all screens of an application as thumbnails, with arrows between them that symbolize available navigation paths. A special form of this kind of storyboards are use case storyboards [Kru99] that are used in the Unified Process [Kru04] to define an abstract high-level description of the user interface belonging to a use case. They are in the form of collaboration diagrams, with boundary objects as placeholders for windows and the like. Also, Apple has included a feature called “Storyboards” into the latest version of its development environment XCode, which allows to define user interface screens and navigation between them in a manner described above.

This kind of visualization is certainly useful to get an overview of the individual screens that an application needs to provide and their relationship to each other. As it does not focus on describing a concrete flow of events of actual use of the system, however, the term *storyboard* does not seem appropriate in this case and should rather be replaced by something that better reflects the intention of its use, for example “navigation map” or the like.

4.4.3 Paper Prototyping

Paper prototyping utilizes the physical medium of paper for the creation of prototypes. The user interface is drawn on sheets of paper in whatever fidelity seems fit for the purpose, ranging from quick pencil sketches to colored, detailed drawings. In order to evaluate a paper prototype, a user sits at a table and gets presented the first screen drawing by a facilitator who takes the role of the computer. The user then interacts with the drawing as if it was a functional user interface by tapping on buttons or writing text in input fields. The facilitator executes the actions of the computer and overlays snippets of images for system output or exchanges the whole drawing in case of a screen change. The facilitator can be supported by additional people during a prototyping session, who look up the currently needed drawings and snippets.

Paper prototypes have the big advantage that they are easy and fast to create. Drafts of a user interface are drawn on sheets of paper and individual user interface elements can be cut out in order to be reused on multiple screens. Depending on the application that is being prototyped, either screens with placeholders for concrete data are created, or partial or complete scenes are drawn that already have some or all instance data embedded in their drawing.

Due to its independence of any electronic tool support, paper prototyping does not require any tool knowledge and does not restrict initial design in cases where interactions are to be defined that are not supported by a tool. Although its appearance may seem too crude to be of any use, even this early draft of a user interface can help in the evaluation of interaction designs and the detection of usability problems [Ret94, Sny03]. The creation of screen drawings via pencil and paper is a fast technique that allows to iterate fast and often, so that different design alternatives can be evaluated with only low creation effort, instead of focusing on a solution too early.

The use of paper prototypes, however, requires at least one facilitator to be present all the time in order to take the role of the computer. As it relies on the physical use of paper, paper prototyping is not easily applied in cases where users are locally distributed. Some study results also report that paper prototyping may not always be the preferred solution as participants might feel uncomfortable [STG03] or rather prefer digital versions of paper prototypes [SB09].

4.4.4 Digital Prototyping

The term *digital prototyping* denotes any form of prototyping that is based on the use of computer support for the creation of a prototype. As mentioned in Section 4.2.3, the fidelity of a prototype can vary in multiple dimensions, for example in the visual appearance or the amount of functionality it offers.

In the special case of a digital prototype whose fidelity is similar to a paper prototype, Arnowitz et al. define it as a *digital interactive prototype* [AAB07]. It offers only limited functionality which makes it easier to create, also for people with limited or no

programming background. In current practice, standard office applications like Microsoft Word or PowerPoint are often used to create digital interactive prototypes, as these applications are often available at hand and designers are accustomed to working with them [AAB07]. The amount of interactions that can be defined with these tools is limited but sufficient for initial user tests. Compared to paper prototypes, digital interactive prototypes have the advantage that the amount of additional work needed to create them is limited and that they can be used for remote demonstrations and user testing, which is not easily possible with paper prototypes.

As development proceeds, more sophisticated prototypes might be created that also require larger programming efforts. In these cases, graphical user interface designers like Google WindowBuilder [Goo12] for Java applications or Apple InterfaceBuilder [App11] for Mac OS X and iOS applications help in the fast creation of user interfaces.

Prototypes that can be created with the SCRIPT framework are similar to digital prototypes, as they have a close resemblance to paper prototypes (see Section 4.4.3) with respect to how they are created, only that they can be executed on an electronic device.

4.4.5 Video Prototyping

Video technology has already been used since end of the 1980s for prototyping of computer systems. Vertelney reports about the use of video at Apple to visualize futuristic user interfaces without requiring large implementation efforts upfront [Ver89]. She describes two different ways of using video: in the first case, the video shows only the prospective user interface and the interactions that take place. In the second case, the video shows both the user interface and the person interacting with the system. This allows to also depict the context in which a user interacts with the system.

Mackay et al. use video during all four phases of their design process [MRJ00]. During *observation*, they create video clips of use scenarios. Then they conduct *video brainstorming sessions*, where new ideas are tried out and videotaped for later reference, with the goal to create many alternative potential solutions. The third phase is *design*, where the ideas from video brainstorming are evaluated and narrowed down to a single solution. In this phase, a video prototype of the developed solution is created. Finally, during the *evaluation* phase, video clips of users interacting with the created system are recorded for later evaluation.

Creighton describes an approach called *Software Cinema*, where video technology is used to create video prototypes of scenarios [Cre05, COB06]. His approach allows to utilize the created video artifacts not only for discussion with stakeholders, but also to use them for later stages of development., which is possible due to an underlying model that connects video artifacts with Live Sequence Charts [DH01].

Video can also be used in participatory design settings, where users act out visionary scenarios of their use of a system, using mockups that are made of foam blocks, for example. While the users present their ideas about how interactions with the system

should take place, they are videotaped and the video recording can then be used for discussions with other users and developers [Bin99].

A special form of video prototyping is virtual video prototyping as presented by Bardram et al. [BBL⁺02]. They use a virtual studio setup that allows to visually combine real actors and props with prototypes of the user interface in real-time. This not only saves time during post processing of the recorded material, but also allows instant evaluation of how well a prototype integrates with its use context. However, the composed image is only visible to the director and not the actors, who need to play their role in an environment where all of the areas that are to be replaced with computer images are in monochrome blue or green. This is necessary so that these areas can be replaced easily later on. As a result, experienced actors need to play the role of the potential users during recording, and the real users can only give feedback when they get to see the final result.

In general, video prototypes follow a scenario-based approach in that they depict a concrete flow of interactions between user and system, but they provide only limited to no interactivity at all, so that they are rather used for demonstration purposes and discussion starters.

4.4.6 Wizard-of-Oz Prototyping

Wizard-of-Oz prototyping [DJA93] is a technique that allows to gather user feedback for a system without actually implementing it. Instead, a hidden facilitator takes the role of the system and responds to user input accordingly. While the concept is similar to paper prototyping, the major difference is that the users are not aware that the reactions of the system are actually triggered by a person instead of the system itself. To them, it looks like they were interacting with the working system. This technique is especially useful for interaction styles that are not yet possible to realize or only with massive development effort, such as speaker-independent, reliable voice recognition.

4.5 Prototyping Tools

During the last two decades, various prototyping tools, especially for the design of user interface prototypes have emerged, which are presented in this section.

In the beginning of the 1990s, the *QUICK* tool and its related *QUID* (*quick user interface design*) method have been developed that allow non-programmers to create graphical systems using direct manipulation of images and a high-level programming language that is easy to learn. The *QUICK* tool is directed at evolutionary prototyping, as it creates code that is supposed to be used for further development [DDN92, ND92].

The *Cooperative Interactive Storyboarding Prototyping (CISP)* tool follows a similar approach and allows to quickly design user interfaces from standard user interface elements and assign them basic functionality. When executing the resulting prototype,

traces of user interaction can be recorded, replayed afterwards and comments from users to certain steps can be captured. The tool has been evaluated in the context of designing a user interface for a VCR [MA93].

A range of tools has been developed based on the idea of creating electronic support for classical paper-based sketching, especially when it comes to the design of user interfaces.

SILK (Sketching Interfaces Like Crazy) is one of the first tools for electronic sketching. It is optimized for use with a pen-like input device and performs real-time recognition of drawn widgets. It allows to add interactivity by defining transitions from one image to another that are triggered when a user clicks on a certain area of an image. [LM95, Lan96, LM01]

DENIM (Design Environment for Navigation and Information Models) follows a similar approach and is geared towards web site designers with minimal or no programming background [LNHL00]. It is also meant to be operated with a pen-style input device in order to minimize the switching barrier for designer who are used to work with sketches on paper. The tool allows to sketch web pages and define transitions between them. It also has a playback component included that allows to evaluate the navigation transitions that have been defined. A zooming mechanism for the drawing canvas allows to change the level of detail the designer wants to work with. At the lowest level, the designer can modify individual elements of a page. At the farthest level, an overview of all pages and the transitions between them is provided. In the initial version of DENIM, transitions could only be triggered by left-clicking. In a later addition [LTL02], a choice between multiple events for triggering a transition, like double-click or timer-based execution of an event, was added.

A tool that is focused on the design of multimedia applications is *DEMAIS (Designing Multimedia Applications with Interactive Storyboards)* [BKC01]. Similar to SILK and DENIM, it supports pen-and-paper style input to comfort designers who prefer this as their favorite way of exploring design alternatives. DEMAIS not only allows designers to create digital sketches, but it also features a basic sketch detection for rectangles so that designers can assign images, video and audio to rectangles that have been detected. Additionally, designers can add text via keyboard, either for visual representation or for narration via text-to-speech, and they can define behavior on sketched elements, depending on user input or progression of time. In order to evaluate the defined behavior, DEMAIS allows to play back the design and use it as a prototype.

All of these tools focus on creating prototypes that allow stakeholders to experience the interaction with a system before it has been realized. None of them, however, establishes a connection between prototypes and other requirements specification artifacts like scenarios, as it is the focus of this dissertation. This missing connection makes it difficult for developers to decide how prototypes and scenarios relate to each other, and bears the risk of inconsistencies when they are developed in a parallel, uncoupled fashion.

4.6 Prototype Knowledge Management

Bäumer et al. identified the problem of making knowledge accessible that is contained in prototypes, which is especially important when prototyping and final implementation are performed by different teams [BBLZ96]. To tackle this problem, different approaches have been developed, which are presented in the following.

Schneider proposes the *FOCUS* strategy [Sch96], which captures information that emerges during the discussion between humans about a prototype. Its focus is on prototypes that have been implemented in an object-oriented language. The discussions that are to be captured take place between the developer of the prototype and another developer who wants to know details about the implementation, and does not involve communication with users. The approach is focused on programmed prototypes, which requires coding skills for creation, while the *SCRIPT* framework focuses on digital prototypes that can be created by people with potentially no programming background, especially designers.

Ravid and Berry try to tackle the problem of knowledge management for prototypes by proposing a six step approach to prototyping [RB00]. The first five steps are concerned with a rather detailed analysis of the context of the system under development and its application domain. The individual characteristics of the system are identified and then the characteristics that are to be prototyped are selected. In the sixth step, the prototype is implemented. While this approach ensures that it is known beforehand which functionality is realized in the prototype, it requires substantial analysis efforts upfront and is thus not suitable for rapidly evaluating different competing design alternatives that should only take minimal effort in order to be easily disposable.

Memmel presents *INSPECTOR* [Mem09], a tool which is mainly focused on corporate user interface development. It supports prototype-driven requirements specification, which means that a prototype is used as a basis for requirements specification, not vice-versa. *INSPECTOR* features a zoom interface and allows to specify the user interface on all levels of visual fidelity, from sketched low fidelity to nearly-polished high fidelity. It follows a unified approach in that it also allows to store arbitrary documents and scenario descriptions, storyboards, and task and role maps. Based on this information, prototypes can be defined and used for evaluation, while simultaneously building up a requirements specification. The *INSPECTOR* tool is similar to the *SCRIPT* framework in that it recognizes the need for prototyping in early stages of development and provides a unified environment to specify prototypes and requirements, including scenarios. The *INSPECTOR* tool, however, only allows to attach textual scenario descriptions to prototypes as a whole, which bears the problem that both need to be kept consistent manually when either artifact is changed. Additionally, the automatic generation of derived documents and videos, as it is possible with the *SCRIPT* framework (see Section 6.6), is not available.

Harel and Marelly developed the *play-in/play-out* approach, which combines prototyping and specifying requirements [HM03]. For play-in, a preliminary version of the

user interface is used by domain experts to specify the desired behavior of the system. This activity is straight-forward for the users in that they operate the user interface by pressing buttons and entering values as if it was already functional, and defining system responses in a similarly direct manner, by directly setting desired output values or visual appearance of elements of the user interface. This activity can also be applied to abstract visual representations of an object model instead of a user interface, where the user triggers methods and sets attributes of objects. The underlying *play-engine* transforms the user input into a formal requirements specification expressed in Life Sequence Charts (LSCs). When the play-engine is set to play-out mode, the preliminary user interface behaves like a prototype, which reacts to user input according to the previously play-in behavior, and hence according to the underlying requirements specification. Similar to the SCRIPT framework, the play-in/play-out approach relies on the use of scenarios, but in this case users specify the system by playing in scenarios with a mockup of the user interface, which are then automatically converted to LSCs. Users of this approach, however, need to have knowledge about LSCs, as they not only interact with the GUI mockups, but also with the LSCs themselves. This is different to the SCRIPT framework, where users are not required to learn about the models underlying scenarios and prototypes. In contrast, the SCRIPT framework represents scenarios as narratives, which do not require knowledge about any formal representation.

Gabrysiak et al. propose an approach which is also called *scenario-based prototyping* [GGS09, GGS11]. Their focus is primarily on the design of multi-user systems in a corporate context. They start with an initial requirements elicitation that leads to an initial process model using the Business Process Model and Notation (BPMN) [Gro12a]. Based on this model, they create a prototype and let users interact with it. Depending on the role a user takes, they are presented different prototype visualizations, according to the activities their role is responsible for as determined in the initial process model. The interactions between user and prototype are not only recorded for evaluation, but also for reuse in subsequent prototyping sessions. For example, consider a process that requires role A to contact role B via email. When a user executes the prototype in role A, all interactions are recorded, including the email the user writes to role B. If the prototype is then executed in role B, the email composed previously by user in role A is reused and presented to the current prototype user, thus making the evaluation of the prototype much more realistic. All data gathered from prototype usage is fed into refinement of the prototype and the underlying process description. Although the approach of Gabrysiak et al. is called “scenario-based prototyping”, it actually operates on the level of abstraction of use cases, which are related to each other via BPMN. Although the prototypes can be generated automatically, the underlying BPMN model needs to be created manually and thus requires knowledge in the use of BPMN. Additionally, the approach only describes how to create generic prototypes to evaluate interactions between roles and does not explain how a prototype with a specific user interface is to be modeled or generated.

Chapter 5

The SCRIPT Model

As presented in the previous chapters, scenarios and prototypes are valuable aids in developing systems with a focus on user-centered design. Narrative scenarios, which are expressed in natural language and using terms of the application domain, provide stakeholders with a concrete description about how interactions with the future system are supposed to take place. As no prior training in formal notations is necessary in order to read a scenario, every stakeholder can contribute to the requirements specification of the system. As explained in Chapter 4, prototypes can be produced for several purposes. With a focus on user-centered design, they are suited best to either communicate the visual appearance of the future system, or to evaluate how the system best fits into a context of use, be it a business process or the users' everyday life.

In this dissertation, the focus is on the latter way of using prototypes, especially during the early stages of software development, where the goal is to explore different alternatives for the design of the user interface, and hence the ways users are supposed to interact with the system. Instead of focusing on a single solution too early, different possible solutions are evaluated with stakeholders to ensure that a best fit can be found. For designers, this requires a willingness to create and also dispose of designs in a quick and iterative manner. Hence, the use of low-fidelity prototypes that can be created without much effort is the preferred approach in this stage of development [Won92, STG03, RSI96, VSK96], which is also recommended by ISO standard 9241-210 [ISO10]. Additionally, the ISO standard recommends to create prototypes that allow stakeholders to solve some realistic task in order to gather sensible feedback.

As has been shown by Rudd et al., the level of fidelity of a prototype influences the kind of feedback that is gathered from users [RSI96]. They found out that low-fidelity representations are more suitable if feedback on general issues is wanted, like the flow of interactions as compared to details of the screen layout. On the other hand, Walker et al. found out that the level of fidelity and the type of presentation (electronically or on paper) did not influence the quality of the feedback they collected [WTL02]. Hence, they promote to use whatever approach fits best into the current state of a project. Especially, they promote to use low-fidelity prototypes, no matter if on paper or electronically, in the early stages of development, where quick iterations and modifications are necessary. As part of the *Usability Engineering Lifecycle*, Mayhew [May99] also promotes the use of

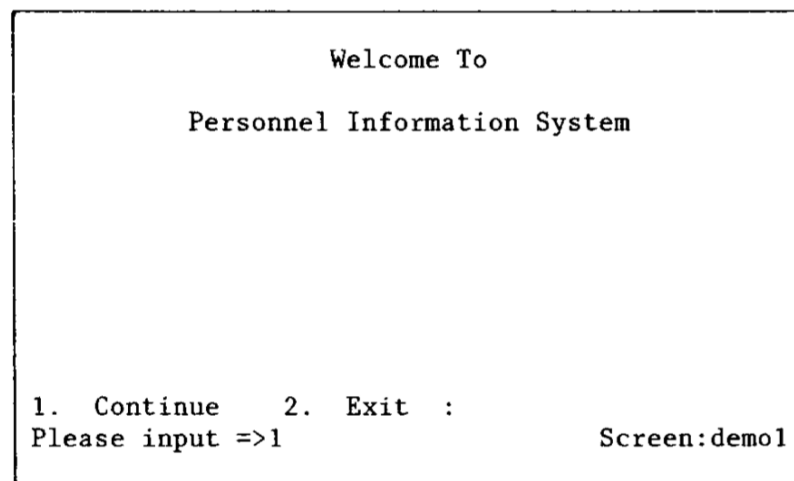


Figure 5.1: Example of a screen generated with the *Screen-Based Scenario Generator*. From [HY88]

low-fidelity prototypes in the beginning of a project in order to ensure that the attention of users is focused on the interaction with the system and not on details about its visual appearance.

Narrative scenarios and prototypes have some major drawbacks if they are used separately. Narrative scenarios need to be enhanced with visualizations, as natural language is not suitable for describing user interfaces. Adding static images alone is not sufficient, as they do not enable users to experience what the interaction with the system will feel like. Prototypes on the other hand can provide the interactivity scenarios are missing. They also allow to evaluate which role a system might take in a given context by handing them to users and letting them gather experience on their own. But more often than not, the interactions that have been realized with a prototype are not documented [Sch96]. Additionally, they cannot convey any information about the context in which a system is used.

In order to exploit the usefulness of scenarios and prototypes to their biggest extent, they should not only be applied side-by-side. The combination of scenarios and prototypes can yield benefits that are greater than their isolated use. Weidenhaupt et al. conducted a study where they analyzed 15 projects regarding the use of scenarios. The results showed that two-thirds of the projects used both scenarios and prototypes, which was beneficial if not crucial to project success according to the interviewed project leads [WPJH98]. The study also revealed that a major problem was to keep scenarios and prototypes in sync, for which no tool support exists yet.

However, the idea of combining scenarios and prototypes is not new. Already in 1988, the *Screen-Based Scenario Generator* [HY88] relied on a combination of scenarios and prototypes. As its prototypes are restricted to terminal-based menu driven programs,

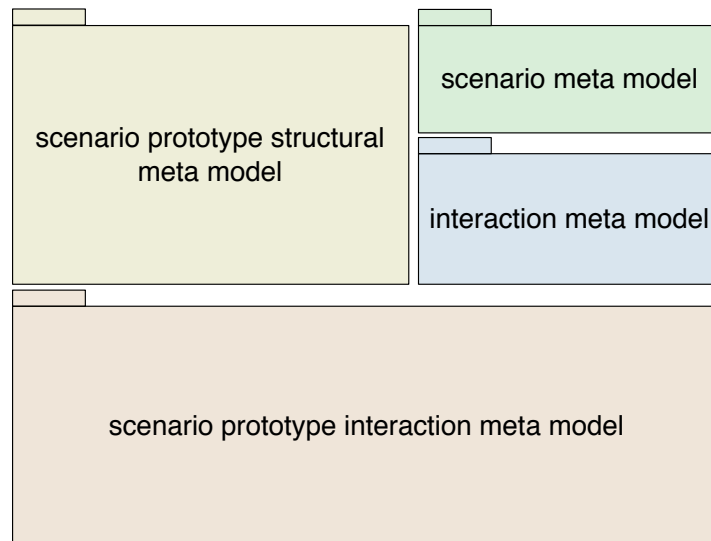


Figure 5.2: Overview of the SCRIPT model, which consists of the scenario meta model, the interaction meta model, the scenario prototype structural meta model and the scenario prototype interaction meta model

as depicted in Figure 5.1, it is unattractive for modern application development. More important, the scenarios that drive the prototypes are not explicitly accessible by the user, but hidden in the prototype logic, which diminishes their usefulness as narrative descriptions of user-system interactions.

The problems described above lead to the development of the **SCRIPT (scenario-driven prototyping)** framework and the model it is based on, which is presented in this chapter. The SCRIPT model provides a meta model for defining scenarios and prototypes, and has been developed iteratively, based on discussions with colleagues, an analysis of available commercial tools in this area, and interviews with professionals who are concerned with design and prototyping [Fel12]. The prototypes that are in the focus of the SCRIPT framework are limited to a specific flow of interactions, namely the one defined by the related scenario. Hence, the prototypes of the SCRIPT framework are called **scenario prototypes**. An overview of the meta models that belong to the SCRIPT model is shown in Figure 5.2.

As scenarios and scenario prototypes that are created using the SCRIPT framework are represented as models, they can be integrated with other requirements specification tools, either by exchange of model elements or by directly integrating tool support for the SCRIPT framework into an existing tool. The latter way of integration is preferred as it ensures that artifacts are kept in one place and that they are directly accessible in later phases of the development.

The chapter is organized as follows: Section 5.1 presents the **scenario meta model** and the **interaction meta model**, which define the structure of a scenario and the interactions

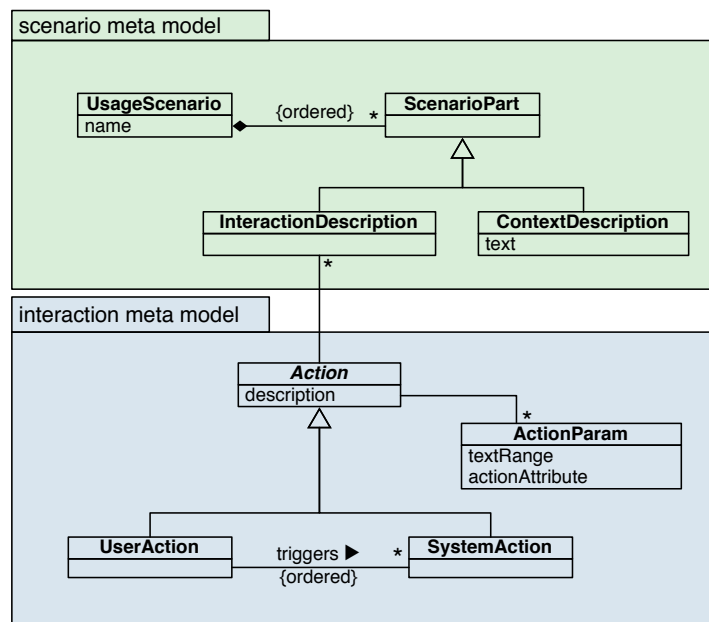


Figure 5.3: Scenario meta model and interaction meta model

described by it. This part of the model is independent of the technology used for building the system and thus applicable to any kind of scenario. These two meta models constitute the first part of the SCRIPT model. The second part of the SCRIPT model allows to define a scenario prototype that enables the user to evaluate the role of the system in its context. This part of the SCRIPT model is realized by the **scenario prototype structural meta model** described in Section 5.2, and the **scenario prototype interaction meta model** described in Section 5.3. As the meta models have relationships defined between each other, changes to artifacts based on one model can be propagated to artifacts based on the other models, thus keeping them consistent. Section 5.4 explains how scenarios and scenario prototypes are related to each other regarding their multiplicity, and Section 5.5 discusses criteria that determine the applicability of the SCRIPT framework.

5.1 Scenario Meta Model and Interaction Meta Model

The first part of the SCRIPT model is targeted at modeling scenarios and the interactions they describe. The focus is on **usage scenarios**, which describe the interactions between user and system on a very detailed level, like entering values in input fields and pressing buttons. This level of detail is necessary to thoroughly evaluate the role of a system, as more abstract descriptions leave too much room for diverging interpretations.

The scenario meta model and interaction meta model can be seen in Figure 5.3. The scenario meta model consists of a `UsageScenario` that has a `name` and acts as a container for the scenario content. It contains multiple `ScenarioParts`, which carry the content of the scenario. A `ScenarioPart` can either serve as a `ContextDescription` or as an `InteractionDescription`. In the former case, it provides information about the context of system usage instead of describing any user–system interaction. This information is stored in the `text` field of a `ContextDescription`. In the latter case, an `InteractionDescription` serves as a bridge to the interaction meta model by referencing an `Action` from the interaction meta model.

The interaction meta model describes concrete interactions between user and system independent of the technology used for realization of the system. In the interaction meta model, the abstract class `Action` provides a textual `description` of the action that takes place. These actions differ in whether a user or the system performs the action, hence two more specific types of `Action` exist, which are `UserAction` and `SystemAction`. For every `UserAction`, the system needs to give some feedback in order to inform the user that it received their input. Otherwise, the user does not know if the system recognized their input and might even think that the system has crashed [Nor02]. Hence, the `UserAction` has an association to the `SystemActions` it triggers and which represent the confirmation of the system that it received the input from the user.

Based on this meta model, the following instantiations regarding `ScenarioParts` and `Actions` are possible, which are depicted in Figure 5.4:

1. A `ScenarioPart` provides information about the context of a scenario. In this case, the `ScenarioPart` is of type `ContextDescription` and holds the relevant information as narrative in its `text` field.
2. A `ScenarioPart` represents an action performed by the user. In this case, the `ScenarioPart` is an instance of type `InteractionDescription` and has an association to a `UserAction`, where the description of the `UserAction` verbally explains what action has been performed by the user. As the system needs to give some feedback that shows that it correctly understood the user action, one or multiple `SystemActions` should be connected to the `UserAction` via the “triggers” association.
3. A `ScenarioPart` represents an action that has been triggered by the system and not as a response to a `UserAction`. The instance of `ScenarioPart` is also of type `InteractionDescription`. This is the case when the system has been activated for some other reasons such as the countdown of a timer, the receipt of an email or the like. While this action might cause a reaction by the user, it does not trigger it like in case 2, as the user is not part of the system.

Note that both the `ScenarioParts` of a `UsageScenario`, as well as the `SystemActions` triggered by a `UserAction` are ordered. Their ordering reflects the chronological order in which the actions take place. The model does not explicitly provide any means

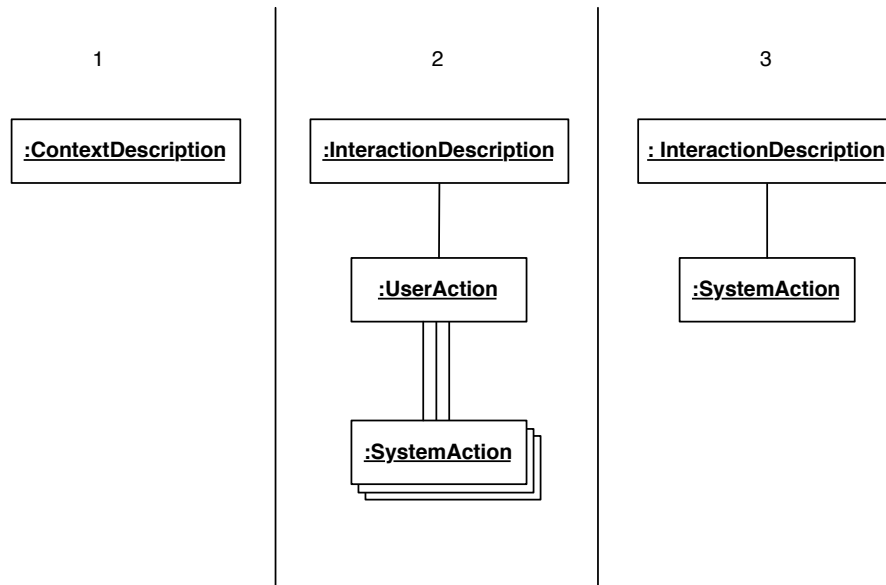


Figure 5.4: Possible instantiations of the scenario meta model and interaction meta model

for expressing parallelism. Should it be necessary to describe events that happen in parallel, where each event starts with some action and ends with some action, they can be converted to a sequential order of actions as follows: Assume two events e_1 and e_2 are supposed to happen in parallel. At the beginning of each event, the system displays to the user that it starts processing, and at the end it displays that it has finished. Hence, each event can be decomposed into a starting `SystemAction` $start_i$ and an ending `SystemAction` end_i , which can then be arranged in sequential order as desired. In case event e_2 happens while event e_1 is running, the resulting sequential order of `SystemActions` would read: $start_1-start_2-end_2-end_1$. As a reminder, the aim of the model is not to provide an exact requirements specification, but rather to provide stakeholders with an idea about how interactions with the system might behave.

The interaction meta model also defines the `ActionParam`, of which each `Action` might have multiple instances. Each `ActionParam` represents a part of the `Action`'s description that is relevant to the execution of that `Action`. As an example, there might be a `UserAction` where a user enters her name into a text field. The description of the `Action` might thus read "She enters 'Lisa' as her name." This sentence contains the information that the value "Lisa" is entered into the system. In order to make this information accessible by specialized subtypes of `Action`, an `ActionParam` can be instantiated. More information about `ActionParams` is given in Section 5.3, where the interactions between user and system are detailed.

In the interaction meta model, no reference to any kind of technology, like voice input or gesture recognition, is made. Scenarios and interactions that are defined with this

model may use any kind of technology, either existing or yet to be developed. The model is only intended to keep information about which interactions take place in which order, but nothing about how they are supposed to be realized. The details about the interactions are stored as plain text in the description field of an Action. This makes it applicable for a broad range of software development.

In order to derive a narrative scenario representation readable for a stakeholder, the texts of all ScenarioParts and descriptions of attached Actions are concatenated. The resulting text shows a close resemblance to a narrative scenario. An editing tool for the scenario meta model and interaction meta model should try to make the duality of model and textual representation as unobtrusive as possible in order to allow system designers to work with it even if they are not familiar with modeling. A description of the prototypical tool developed in the scope of this dissertation is given in Chapter 7.

5.2 Scenario Prototype Structural Meta Model

The second part of the SCRIPT model is concerned with modeling scenario prototypes and the interactions that are defined by them.

As could be seen in Chapter 4, prototypes can be built for many different purposes. The prototypes that are supported by the SCRIPT model are strongly related to paper prototypes, only that they exist digitally. In order to get a better understanding of the components that (paper) prototypes are made of, some terminology is defined upfront:

A **screen** defines the static structure of a coherent set of user interface elements. For smaller mobile devices like smartphones, this typically corresponds to the whole content of the display. On larger mobile devices like tablets and desktop-sized computers, this typically corresponds to a window. Note that the notion of a screen is only concerned with the static structure, not with the content that is displayed. For example, when considering a typical mail application like Mozilla Thunderbird or Microsoft Outlook, the screen consists of a left column showing the available mailboxes, an upper right part that shows all available mails and a lower right part that shows the content of a selected mail. As long as this layout does not change, the screen stays the same, no matter which mailbox or mail is selected.

A **scene** is an instance of a screen, which also takes into account the content that is being displayed. Thus, regarding the mail application example, if a different mail is selected so that the content of the lower right part changes, this would constitute a new scene, while the underlying screen stays the same.

The scenario prototype meta model is based on the concepts of paper prototyping as described in Section 4.4.3. A notable limitation of paper prototypes is their restricted ability to react to arbitrary user interaction. As all scenes that are supposed to be presented to the user (that is, the screens with the concrete data to be displayed) need to be prepared in advance, only a small set of possible interactions can be realized with this kind of prototyping. Another drawback of paper prototyping comes with the

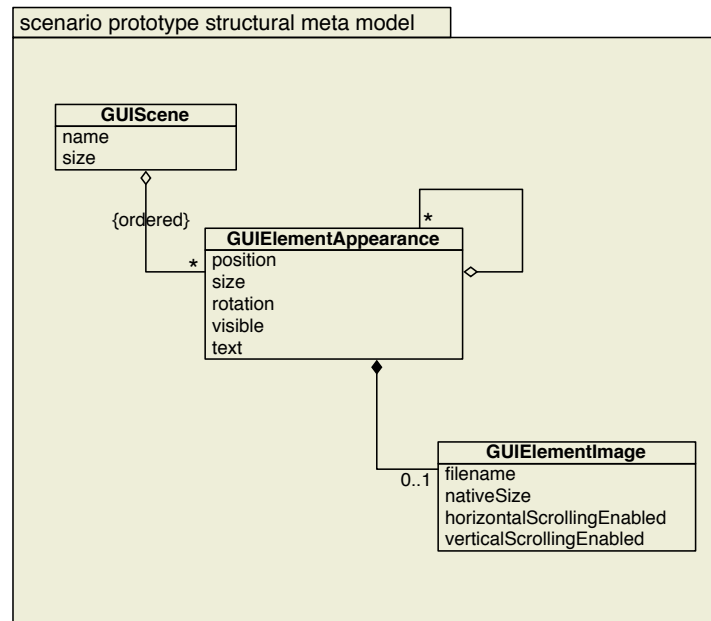


Figure 5.5: Scenario prototype structural meta model

medium is uses: paper. Depending on the set of interactions that should be available with a paper prototype, the amount of paper sheets and snippets that need to be kept around during execution can grow huge and make the fluid operation of the prototype a challenging task for the facilitator, who takes the role of the computer. Additionally, it is not easy to use paper prototyping in order to evaluate mobile applications in their real context. Finally, when a prototype design has been decided on for further refinement and development, the paper prototype and the knowledge it contains are not easily accessible for a developer, as they are neither digitally available nor connected to other development artifacts.

The scenario prototype meta model provides the basis for a digital equivalent to a paper prototype, and hence focuses on two-dimensional graphical user interfaces (GUIs), compensating for the shortcomings mentioned above. Prototypes which are based on this model are called scenario prototype, since they enable users to experience a concrete scenario. An important aspect in the creation of the meta model was to keep it as generic as possible in order to support a large number of applications, without constraining it unnecessarily to a specific platform. Additionally, any tool support for working with the meta model should be able to completely hide the model from the user and allow them to create prototypes as easily as if they were doing classical paper prototyping. This is important as to retain a spirit similar to sketching on paper, i.e. trying out various design alternatives instead of narrowing down on one solution prematurely. In order to maximize the acceptance of the SCRIPT framework, it needs to be usable without requiring users to learn about or actively conform to the underlying model.

The scenario prototype structural meta model is shown in Figure 5.5. The basic units of a scenario prototype are scenes that are displayed to the user. These scenes are called `GUIScene` in the meta model. A scene consists of the structure of the user interface as well as the concrete data that is displayed, and thus resembles a single sheet in classical paper prototyping. It has a `name` and a `size`, which can change from scene to scene. One example of a changing scene size is a prototype for a mobile device, where the orientation of the device changes between two scenes. This would be represented by two scenes where width and height have been swapped.

A `GUIScene` consists of multiple regions that allow users to interact with the prototype. Such a region is called `GUIElementAppearance`, as it corresponds to a representation of a user interface element. The ordering of `GUIElementAppearance`s is important as it defines which regions are located above others in case their borders are overlapping. This concept is similar to layers in graphics applications. Note that a `GUIElementAppearance` does not necessarily have a visual representation of its own; in many cases, it simply marks a region on the underlying graphic a user can interact with. Each `GUIElementAppearance` has a `position` and `size` that defines the “hot zone” where a user interaction can take place. It also has a `rotation` and a flag defining if it is `visible` or not. The latter is of interest for defining sequences of interactions. A `GUIElementAppearance` can also have a `text` defined with it.

As stated above, a `GUIElementAppearance` does not need to have an image of its own, in which case it simply marks a region on the underlying graphic that allows for user interaction. In case it is supposed to have a visual representation, it is linked to an instance of a `GUIElementImage`. This object stores information about the file containing the image data and the native size of the image. By default, an image is scaled to fit the size defined by its `GUIElementAppearance`. In order to work with long lists and other elements that do not fit into a scene completely, horizontal and vertical scrolling can be enabled separately for a `GUIElementImage`. For example, consider the region constituting a `GUIElementAppearance` and the image of a list shown in the left part of Figure 5.6. If no scrolling mode was set, then the image would be displayed as in part (a). If `verticalScrolling` was enabled, then the image would be displayed as in part (b). The lower dotted gray part of the image in part (b) would not be visible, but instead the user could scroll inside the region defined by the `GUIElementAppearance` in order to access the lower part of the image.

A `GUIElementAppearance` can also have a `text` defined with it. This is because with classical paper prototyping, every text that is shown to the user either needs to be prepared as a separate piece of paper with the text on it, or the text needs to be written onto the underlying picture during the prototyping session. For a scenario prototype, this means that every text to be displayed would need to be available as a separate image, which causes a lot of additional work for the designer of the prototype. As entering and displaying text is a common interaction on GUIs, the model allows to specify a `text` for a `GUIElementAppearance` that should be displayed inside the region it defines. This is not only relevant for the static model of a scenario prototype, but especially when it

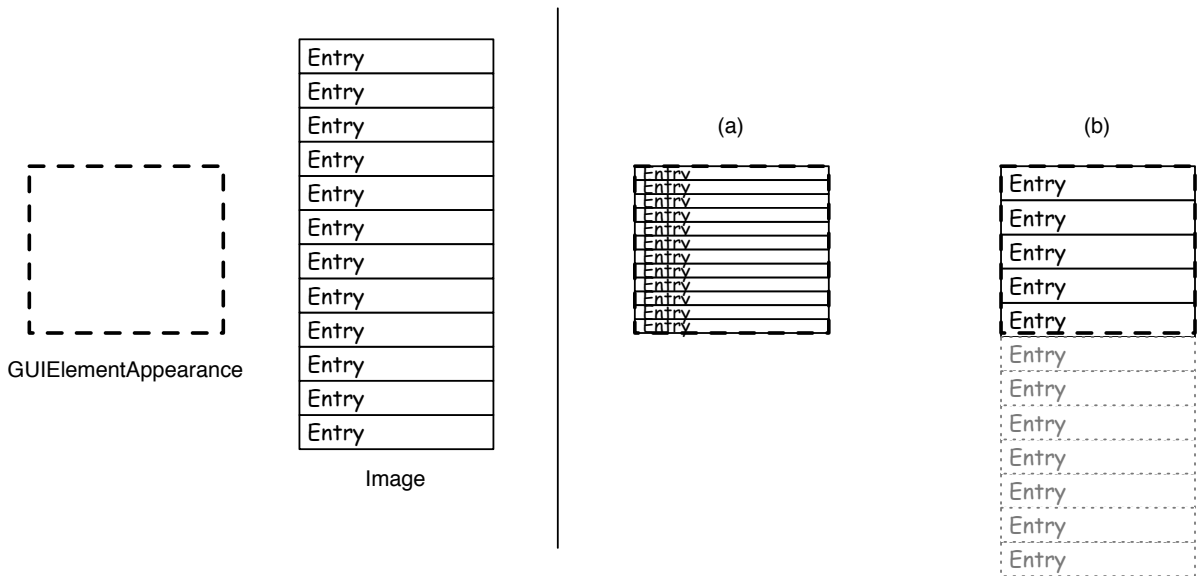


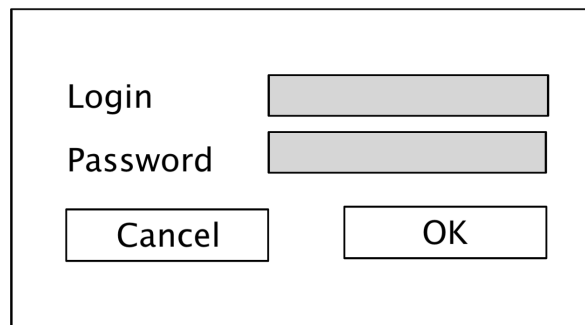
Figure 5.6: Example of `GUIElementImage` without (a) and with (b) scrolling enabled

comes to specifying the interactions between user and system that are described in the next section.

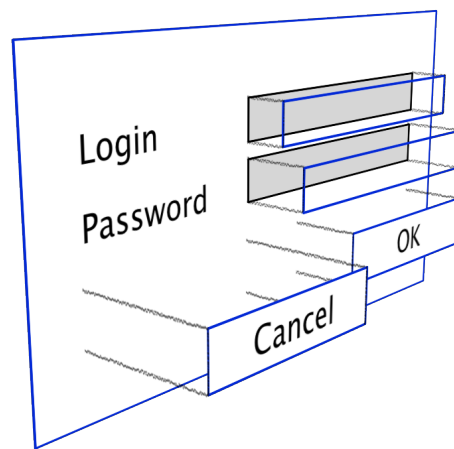
As can be seen by the aggregation of `GUIElementAppearance` to itself, it is possible to nest `GUIElementAppearances` inside each other. The child objects are only visible and active when their parent objects are visible and active. In case a `GUIElementAppearance` has a `GUIElementImage` and at least one type of scrolling is active, child `GUIElementAppearances` can also be located on the protruding part of the image. For example, on the list shown in Figure 5.6 (b), it is possible to create a child `GUIElementAppearance` on the lowest item of the list. In this case, the user of the prototype had to scroll down the list in order to activate it.

Typically, a `GUIScene` consists of at least one `GUIElementAppearance` that fills the whole space of the `GUIScene` and that has an image attached to it. This image constitutes the background image of the `GUIScene`. Multiple `GUIElementAppearances` can be located “on top” of it, i.e. in layers above it, which means they are positioned further behind in the list of `GUIElementAppearances` of the `GUIScene`. These additional `GUIElementAppearances` either bring their own visual representation that adds up to the resulting scene that is displayed to the user, or they have no image attached, in which case they mark regions on the background image users can interact with.

As an example, consider the case of a simple login dialog displayed in Figure 5.7. Part (a) shows the final `GUIScene` presented to the user. Part (b) displays how this scene is composed of single `GUIElementAppearances`, which are recognizable by their blue border. On the lowest level, a `GUIElementAppearance` is positioned that has an image which contains the texts “Login” and “Password”, and areas for entering text.



(a)



(b)

Figure 5.7: Example of layering in a GUIScene. Part (a) shows the resulting scene, part (b) shows the division into single GUIElementAppearances

Note that these are part of the image and provide no functionality by themselves. In order to allow the user to click on the text fields and enter text, the upper two GUIElementAppearances are added that have the same size as the text fields, but no image of their own. This way, interactivity can be added to otherwise static images. The “Cancel” button and “OK” button are not part of the underlying graphic and thus two GUIElementAppearances are added that have an image of their own. All images that are defined via GUIElementAppearances are merged when a GUIScene is presented to the user.

5.3 Scenario Prototype Interaction Meta Model

This section describes the concrete UserActions and SystemActions that are defined for a scenario prototype. As a reminder, a scenario prototype is only concerned with the interactions between user and system, not the internal operations of the system. Hence,

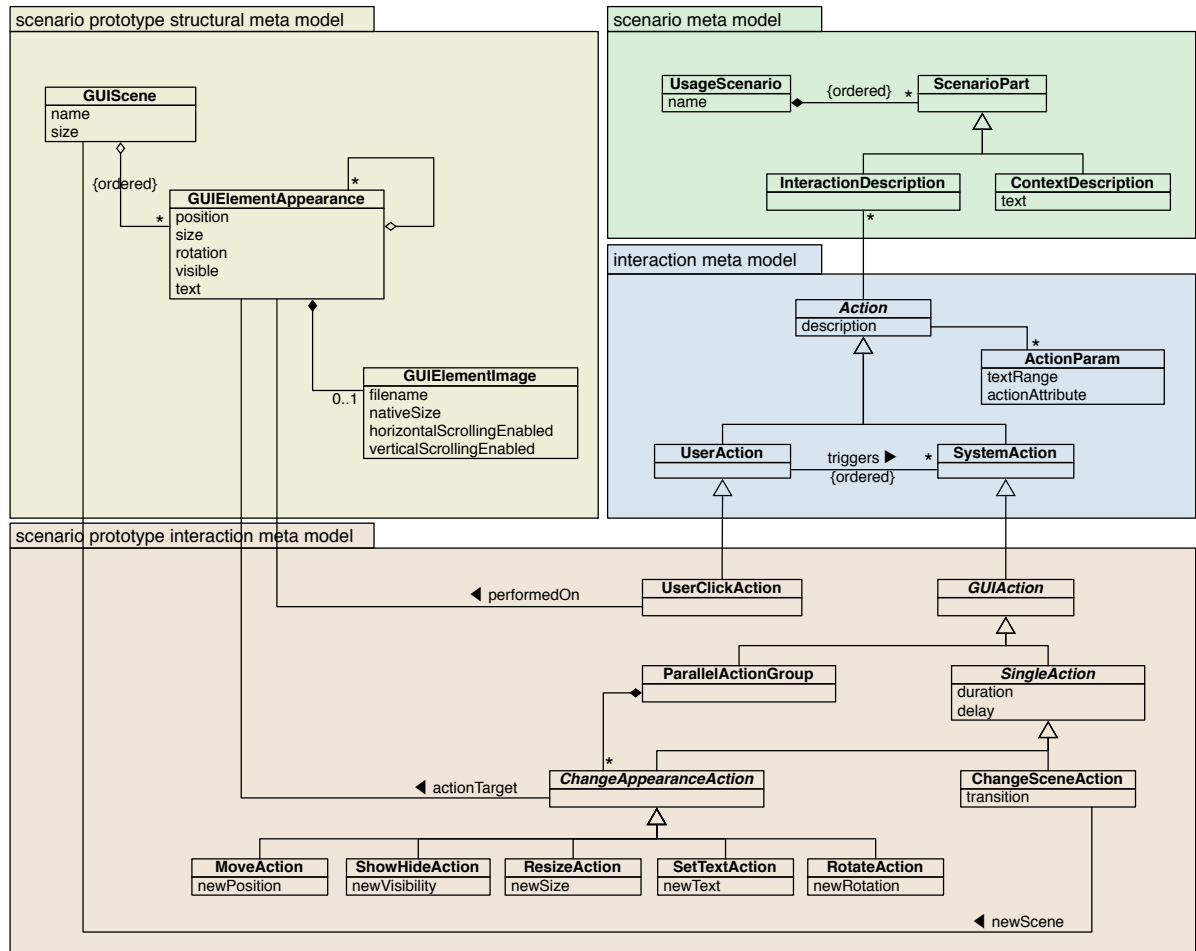


Figure 5.8: Complete model of SCRIPT

the scenario prototype interaction meta model is restricted to modeling the behavior of the GUI.

The complete SCRIPT model can be seen in Figure 5.8. For the scope of this dissertation, only a `UserClickAction` has been defined in the scenario prototype interaction meta model as concrete type of `UserAction`, linking to the `GUIElementAppearance` on which the action can be performed. While the restriction to `UserClick` is certainly a limitation, to click (or to tap on a device with touch input) is still one of the most important user actions on a GUI, and other user actions can often be substituted by clicking. For example, in a system that is operated with a touch interface, instead of allowing the user to swipe on an element, a small arrow can provide the same functionality when clicked. Although the *feel* of the interaction is different than in the final system, the evaluation of the *role* of the system is not compromised, regarding the classification of prototypes

by Houde and Hill (see Section 4.2.4). However, the scenario prototype interaction meta model is intended to be extended to support other user actions as well.

The case of a user entering data requires special attention. The scenario prototype follows the sequence of actions that are defined by a scenario, thus only the data defined in the scenario can be entered. Whenever users enter data into a system, they need to be able to see if the system has recognized their input. In the case of textual input via keyboard, typically the entered characters are echoed on screen. In terms of the scenario prototype meta model, the action of a user entering data is realized as a combination of a `UserAction`, that represents the user clicking on a `GUIElementAppearance` where they want to enter some data, and a resulting `SystemAction`, that defines how the system displays the data that the user entered, as defined in the underlying scenario. In other words, the system skips the details of data input by the user and immediately displays the result to the user, i.e. the entered data. In case of a mobile application, where data is typically entered with an onscreen keyboard, an intermediate step can be defined where the image of an onscreen keyboard appears, and a click on the keyboard results in the data being filled in automatically. Though possible, it is normally not useful to define a prototype that requires the user to enter each and every character manually, as the text that can be entered is predefined in the scenario and most users are already familiar with the concept of a keyboard.

The main part of the scenario prototype interaction meta model is dedicated to the possibilities for the system to react to a `UserAction`. This model is focused on 2D GUIs, but it is possible to extend the model in order to build prototypes with, for example audible `SystemActions` like playing an alarm sound or speech output, by subclassing `SystemAction` accordingly. Each `SystemAction` can either be realized by a `SingleAction` or a group of `ChangeAppearanceActions` that take place in parallel, as described below. A `SingleAction` defines a `duration` and a `delay` for its execution.

Two types of `SingleActions` can be distinguished: actions that operate on a `GUIElementAppearance`, and actions that change the currently visible scene. For a `ChangeSceneAction`, a `transition` can be specified that describes if and how the change of scenes should be animated. As the available transition types depend on the platform the prototype is developed for, they are not specified in the model, but can instead be stored with textual identifiers, e.g. “flip”.

The other type of `SingleAction` is the `ChangeAppearanceAction`. It is performed on a targeted `GUIElementAppearance`, which can also be the one that triggered the action. `ChangeAppearanceActions` can be further refined into actions that modify single parameters of a `GUIElementAppearance`. For example, a `ShowHideAction` results in a `GUIElementAppearance` to be displayed or hidden once the action is performed. This is especially useful if the targeted `GUIElementAppearance` has an image attached, so that the image appears or disappears on the user interface of the scenario prototype.

Multiple sequential animations can be realized by using multiple `SystemActions`, each corresponding to a `SingleAction` or a `ParallelActionGroup`, which can contain multiple `ChangeAppearanceActions`. The delay parameter inherited from `SingleAction` can be

used to precisely coordinate multiple `ChangeAppearanceActions` that are grouped in a `ParallelActionGroup`.

As mentioned in Section 5.1, the `ActionParam` in the interaction meta model allows to access values that are defined in the description of a scenario. This is mostly of interest in combination with the `SetTextAction`. Consider again the example of a `UserAction` where a user enters her name into a text field. Assume the description of the Action reads “She enters ‘Lisa’ as her name.” Instead of setting the attribute `newText` of the `SetTextAction` manually to the value “Lisa”, the `SetTextAction` can refer to an `ActionParam` that identifies the part of the scenario text that contains the value to be used in the action. Thus, changes in the scenario description get immediately reflected in the scenario prototype.

The combination of all four models allows to simultaneously develop scenarios and scenario prototypes while ensuring that both stay consistent. Each user–system interaction of a scenario is represented by an `InteractionDescription` and is linked with either a `UserAction` or a `SystemAction` of a scenario prototype, which adds interactivity to it. Conversely, the `InteractionDescription` connects each Action in a scenario prototype to a scenario, which provides context information with its `ContextDescriptions`. These connections enable developers to navigate between scenarios and prototypes, and allow tools to automatically check for consistency. Additionally, the meta model for scenarios and scenario prototypes makes them accessible to developers in subsequent development steps on a fine-grained level.

5.4 Relationship between Scenario and Scenario Prototype

A prototype in the general sense enables users to experience an arbitrary number of user–system interactions. It can allow users to enter arbitrary data, either without actually influencing how the flow of events in the prototype proceeds, or it contains some business logic that correctly responds to the user input. The latter type of prototype most certainly requires some programming effort in order for such a prototype to be realized.

The type of prototypes that are in focus of the SCRIPT framework are scenario prototypes. These prototypes can be classified in that they do not allow arbitrary user input, but only data that has been predefined in a related scenario can be entered during the execution of the prototype. Still, a single scenario prototype might allow several scenarios to be traversed. To some extent, this can also be realized with the scenario prototype model presented in this dissertation, as described next.

In order to construct a deterministic scenario prototype model, each `GUIElementAppearance` must only be related to one instance of a specific subclass of `UserAction`. Otherwise, e.g. if two `UserClickActions` were related to one `GUIElementAppearance`, it

would be undefined which `UserClickAction` is activated once a user clicks on the `GUIElementAppearance`. However, a `UserClickAction` might relate to multiple `ScenarioParts` that belong to different scenarios, e.g. scenarios that describe normal and exceptional behavior, where certain interactions overlap. If multiple subclasses of `UserAction` are available, they can be used to specify different system responses to `UserActions` on a `GUIElementAppearance`, depending on the type of `UserAction`. These can also be part of different scenarios.

Regarding `GUIScenes`, it is possible to reuse a `GUIScene` for multiple scenarios if different `GUIElementAppearances` of the `GUIScene` are relevant for the different scenarios. For example, consider a `GUIScene` that depicts a main menu with three options. In one scenario, the first option is selected, and in another scenario the third option is selected. In this situation, the same `GUIScene` might be used for both scenarios, with two `GUIElementAppearances`, one for the first option and one for the third. Each of the `GUIElementAppearances` specifies the system reactions as defined by their respective scenario. This kind of `GUIScene` reuse might of course also be relevant for the sequence of interactions of a single scenario.

Although possible, the representation of multiple scenarios with a single scenario prototype is not recommended, as it bears the potential risk of unintentionally making changes to interactions of one scenario while editing another scenario. The apparent advantage of having to change less `GUIScenes` when adjustments are necessary can easily become a disadvantage. When it comes to evaluate different design alternatives, introducing high coupling too early in the beginning of a project can hinder the exploration of the design space.

5.5 Criteria of Applicability

As has been presented in Chapter 4, the field of prototyping is very broad. Instead of trying to support all possible kinds of prototyping, the `SCRIPT` framework has been developed to support a focused set of prototyping situations. In the following, the criteria that influence the applicability of the `SCRIPT` framework are discussed.

5.5.1 Platforms

In order to determine the applicability of the `SCRIPT` framework for a given platform, a distinction has to be made between the scenario meta model and interaction meta model on the one hand, and the scenario prototype meta models on the other hand.

As described before, the scenario meta model and interaction meta model make no reference to any technology used for realizing interactions. Their only focus is on structuring the interactions that take place between user and system, and they do not model the way how these interaction are realized. Their use is thus not limited to any technology, like a certain platform or interaction technology.

In contrast, the scenario prototype meta models that are presented in this dissertation have a clear focus on the development of two-dimensional graphical user interfaces (2D GUI). They are concerned with the description of prototypes that allow users to experience the interaction with a system. Hence they can be applied for evaluating any kind of system that utilizes a 2D GUI, independent of the operating system or hardware platform. Potential types of systems reach from classical desktop applications, to applications for mobile devices such as mobile phones and tablets, to embedded devices with a graphical user interface, assuming their ability to play back the prototype in order to evaluate it in its destined context.

5.5.2 Modes of Interaction

Regarding the interaction meta model, virtually any type of user–system interaction can be expressed, as the interaction meta model only defines an abstract `UserAction`. In the description of the action, any existing or novel interaction mode can be described. When it comes to the definition of an according prototype, a textual description is no longer sufficient, but instead subclasses of `UserAction` need to be defined in order to derive a prototype from the model.

For the scope of this dissertation, only a `UserClickAction` has been defined in the scenario prototype interaction meta model as action that can be performed by a user. As already argued in Section 5.3, while this is certainly a limitation, to click (or to tap on a device with touch input) is still one of the most important user actions on a GUI, and other user actions can often be substituted by clicking. For example, in a system that is operated with a touch interface, instead of allowing the user to swipe¹ on an element, a small arrow can provide the same functionality when tapped. Although the *feel* of the interaction is different than in the final system, the evaluation of the *role* of the system is not compromised, regarding the classification of prototypes by Houde and Hill (see Section 4.2.4).

However, it is possible to extend the scenario prototype interaction meta model with additional ways for users to interact with a prototype. For example, a `UserSwipeAction` could be defined, with specializations depending on the direction of the swipe.

When it comes to evaluate interaction modes that require more sophisticated interaction patterns like speech input, the scenario prototype meta model is no longer suitable. In these cases, other prototyping techniques, which most likely also require some amount of manual programming, should be chosen.

¹A swipe is an interaction, typically on a device with touch input, where the user puts their finger on the device and moves it into a direction while simultaneously lifting the finger up, as if giving the underlying GUI element a drift.

5.5.3 Degree of User Interface Content Change

The interactivity of a scenario prototype is defined by UserActions and the resulting SystemActions. In the scenario prototype interaction meta model presented in this dissertation, the possible UserActions have been restricted to UserClick, and the available SystemActions comprise moving, rotating and scaling GUIElementAppearances, making them visible or invisible, and setting the text that is displayed by them. While these actions already allow to express many user–system interactions, still not all kind of systems can be prototyped.

This is especially true for systems with a high degree of user interface content change. This means that an important part of the system functionality results in a frequent change of large portions of the user interface. Typical examples of such systems are games, especially realtime games that allow to move in a three-dimensional space, and software like drawing tools, whose main focus is on the manipulation of a canvas-like interface. While it might still be possible to prototype certain aspects of these kinds of systems, the designer needs to consider if the simplifications that have to be made too strongly limit the usefulness of the prototype.

5.5.4 Amount of User–System Interaction

The amount of user-system interaction is not only relevant for the SCRIPT framework, but for all prototyping approaches that aim at evaluating the role of a system in its context. For systems with only a very small amount of user–system interactions regarding frequency and complexity, the application of *role* prototyping (see Section 4.2.4) might not be necessary. Instead, once the small set of interactions has been identified, additional effort should be invested to make those interactions as user-friendly as possible.

However, as soon as the interactions between user and system happen more frequent and/or consist of more than only a very simple flow of interactions, it is strongly recommended to apply the SCRIPT framework from the very beginning of development.

Chapter 6

Application of SCRIPT

In the previous chapter, the models of the SCRIPT framework have been described. This chapter deals with the details of applying the SCRIPT framework in the context of a software development project.

The chapter is organized as follows: Section 6.1 describes the activities that are involved in applying the SCRIPT framework in a development effort. Section 6.2 explains how the SCRIPT framework can be integrated into existing software development life-cycles. Section 6.3 describes two ways in which the SCRIPT models can be traversed. Section 6.4 discusses potential sources of graphics for use in scenario prototypes and their suitability regarding the intention of the SCRIPT approach. Section 6.5 explains how scenarios and scenario prototypes that result from application of the SCRIPT framework can be used for subsequent requirements specification activities. Section 6.6 shows how static documents, such as textual descriptions and storyboards, as well as videos can be automatically generated from the SCRIPT models.

6.1 Activities in the SCRIPT Framework

The activities that belong to the SCRIPT framework are part of the requirements elicitation phase (see Section 2.2). The activities that are involved when applying the SCRIPT framework can be seen in Figure 6.1 as a UML activity diagram. The swim lanes depict which activities are to be executed by which role. Activities that are located on the border of a swim lane require participation of multiple roles.

The SCRIPT framework involves two main roles: that of a designer and that of an user. Note that both roles may be filled by multiple persons. Designers are concerned with building the system, more specifically with defining the flow of interactions and creating a first draft of the user interface, while users are eventually going to use the system once it has been developed. They possess knowledge about the application domain and current work practices on a very detailed, operational level. They must not be confused with other stakeholders like clients, who fund the project, but often do not have such detailed knowledge about work practices. The success of a project may well depend on the ability to get access to real users, while often it might not be easy to achieve this.

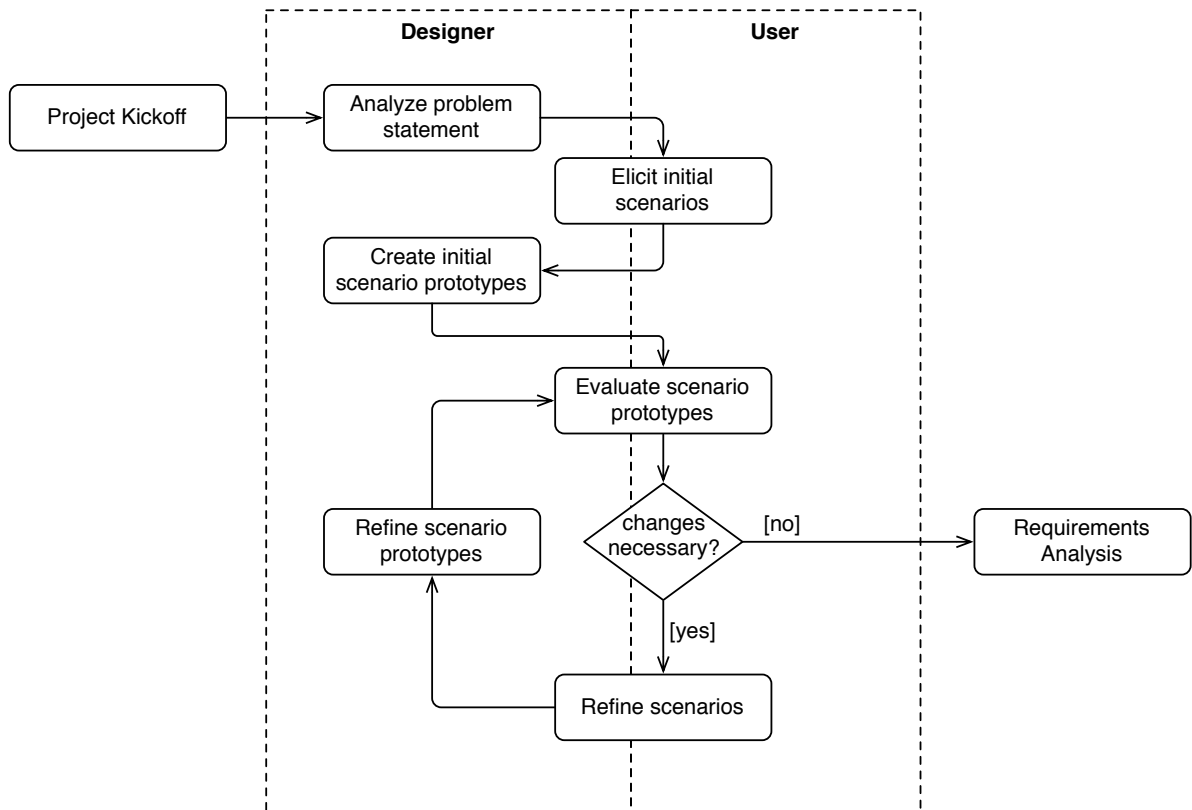


Figure 6.1: Overview of the activities involved for applying the SCRIPT framework

The designer starts by analyzing the problem statement that has been provided by the client in order to get a first impression about the general direction of the project. This includes a first definition of the set of users involved and the tasks they need to perform with the system. In the next step, designer and users come together to derive an initial set of scenarios. Standard scenario elicitation techniques as described in Section 3.3 can be used for this activity. As usual, focus should be set on covering both typical scenarios as well as examples of exceptional, but likely scenarios.

After that, it is the task of the designer to create an initial set of scenario prototypes according to the developed scenarios. Once they are finished, scenarios and scenario prototypes get evaluated by the users. If the evaluation shows that either a scenario or a scenario prototype requires changes, first the scenarios are reviewed together with the users. When designer and users agree on the state of the scenarios, the designer adjusts the scenario prototypes in order to reflect the changes. Note that the decision point “changes necessary?” also refers to the need for new scenarios or the deletion of existing scenarios. In the latter case, it is advised not to completely dispose of the already defined scenarios and scenario prototypes, but rather to keep them and mark them as no longer relevant, which might change at a later time.

Once the set of scenarios and scenario prototypes requires no more changes, so that the decision point “changes necessary?” can be answered with “no”, development proceeds and the resulting artifacts can be used in subsequent development activities like requirements analysis.

6.2 SCRIPT in Development Lifecycles

The activities of the SCRIPT framework are designed to be part of a development lifecycle that eventually results in delivering a running system.

The SCRIPT framework has the advantage that it is not intrusive with respect to the development lifecycle. It consists of requirements elicitation activities that only require the underlying lifecycle to have a phase for developing system requirements. As can be seen in Figure 6.1, the SCRIPT framework in itself allows iteration. This however does not limit its applicability for iterative development lifecycles. Even in a mostly sequential lifecycle, the SCRIPT framework could well be applied, as it does not enforce iterations crossing the boundary of the requirements phase.

These properties makes it just as suitable for any iterative and incremental development lifecycle as well. The description of the SCRIPT activities in Section 6.1 did not mention anything about completeness of the set of scenarios and scenario prototypes. This is by intention, as it is up to the underlying development lifecycle to decide when the critical mass for proceeding to the next development activity has been reached. This allows to build systems iteratively and incrementally, adding additional scenarios in every iteration.

Regarding the development methodology, however, the SCRIPT framework does make some assumptions. It requires the willingness to apply user-centered design (UCD) techniques, most importantly scenarios and prototyping, which form its basic principles. The SCRIPT framework can be considered a UCD method itself. However, it tries to bridge the gap between UCD and software engineering by providing a model that makes the resulting artifacts accessible for later phases of the development.

6.3 Sequence of Model Traversal

The activities presented in the Section 6.1 suggest a use of the SCRIPT model that starts with the scenario meta model and interaction meta model for modeling scenarios, and then moves on to the scenario prototype structural meta model and interaction meta model for defining the related prototype. This corresponds to the path labeled (a) in Figure 6.2.

Another sequence of model traversal is also possible. In this case, a designer decides to first build (parts of) a scenario prototype and defines the underlying scenarios afterwards. The path labeled (b) in Figure 6.2 depicts this situation. This approach can be useful

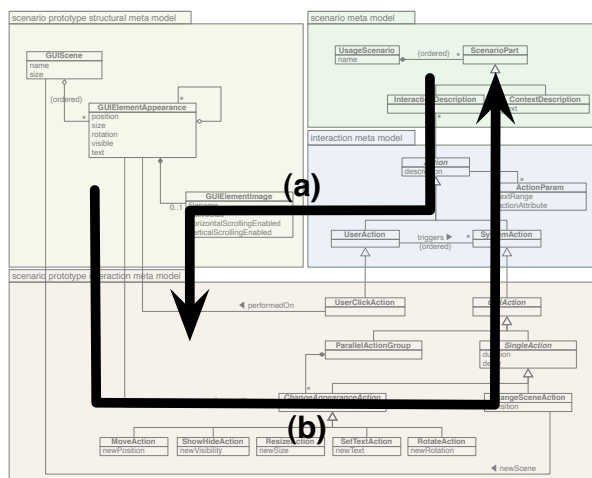


Figure 6.2: Orders of SCRIPT model traversal

in situations where a first meeting with users already took place, so that the designer already has a general understanding of the system context and wants to quickly try out alternative solutions. The choice of which path to follow does not need to be made upfront, but can be changed depending on the current situation.

6.4 Graphical Input for SCRIPT

Depending on the needs of the development project, the graphics used for scenario prototypes can come from different sources. Independent of where the graphics come from, it is important that they still communicate a “work-in-progress” attitude to the user of the scenario prototype. If the graphics, and hence the resulting scenario prototype, already bear a polished and finished look, feedback from users may no longer tackle general workflow issues, but they rather focus on details of the GUI design like layout and choice of color or fonts. While this is valuable feedback during later stages of development, in the beginning of a project the primary goal is to validate the flow of interactions of a scenario. Therefore, working with images that underline the fluidity of the details of the GUI design stimulate focusing on the relevant parts of a scenario prototype, namely the interactions it defines. Suitable for this kind of focusing are images that bear a sketchy look, as if they had been quickly scribbled on a piece of paper and are thus welcome to be altered or disposed of.

The scenario prototype structural meta model has no notion of standard user interface concepts like windows, buttons, text boxes and the like. Instead, it only knows about the abstract concepts of `GUIScene`s and the `GUIElementAppearance`s they contain. This might seem like a limitation, as prototyping tools exist that provide the same user interface elements that are used during implementation later on [MA93, HM03]. These

tools let designers create user interfaces by selecting standard elements from a palette and arrange them on screen. Most of these tools provide export functionalities that not only allow to create images of the user interface, but also to directly export them to a development tool or even directly to code.

This apparent drawback of the scenario prototype structural meta model is in fact one of its strengths. As it is not restricted to existing user interface libraries, it is applicable for the design of arbitrary user interfaces, either using standard user interface elements or designing non-standard, highly specialized designs where every screen is designed individually. For the design of standard interfaces, it is possible to build an editor that refines the scenario prototype structural meta model and provides a palette of standard user interface elements, so that the resulting GUIs can be directly reused during development.

6.4.1 Paper-Based Sketching

As with classical paper prototyping, graphics can be created with pen and paper and be digitized afterwards for use in a scenario prototype. Many current mobile phones feature integrated cameras with a resolution high enough to adequately capture images which can be used for a scenario prototype. In case such a device or a similar one is available, no additional hardware is necessary for digitizing “analog” designs. However, the digital images may need some post-processing, as they need to be cropped to the right size for use in a scenario prototype. In case they were photographed maybe even more sophisticated correction of the perspective is necessary in case the photograph was not taken completely perpendicular to the supporting underlay.

Some of this work like correction of the perspective can be automated. Sketching sheets can be printed that contain special markers. Once they are photographed, an algorithm detects the markers and derives the necessary perspective corrections. For the design of GUIScenes, tool support can even go further. In case the size of scenes for the target platform is predefined, as it is the case with mobile devices where the whole display size should be used, sketching sheets with the previously mentioned markers and placeholders can be printed. Once they are photographed, an algorithm can not only derive the necessary perspective corrections, but also automatically crop the design to the right size. Such an approach has the benefit that a designer can focus on the creation of design alternatives completely independent of any technological restrictions. Such a tool support ensures that the follow up effort for creating a scenario prototype is kept at a minimum.

6.4.2 Digital Sketching

As described in Section 4.5, a broad range of tools exist that allow designers to create sketches using electronic support. The resulting images are typically very similar to sketches made with pen and paper and are thus just as suitable for use with the SCRIPT

framework. Some digital sketching tools are geared towards the design of user interfaces, which sometimes also include functionality to recognize the elements drawn by a designer and replace them with standard user interface elements. This kind of automatism should be treated with caution. If it results in the designs looking too finished, such a tool should not be used or the automatism should be turned off if possible in order to retain the “work-in-progress” character.

6.4.3 Building from Predefined Shapes

Many digital design tools allow to create drawings based on predefined shapes that are available via palettes. Designers can take arbitrary numbers of those predefined shapes, place them on their drawings and modify them to some extent regarding size, color, etc. These tools can be divided into two subgroups: tools that are solely concerned with creating images, and tools that are part of an integrated development environment (IDE). Tools from the former category typically have more flexibility concerning the type of drawings that can be created and also allow arbitrary shapes to be created. Typical representatives of this category are Microsoft Visio or OmniGraffle by The Omni Group. These tools can be extended by adding custom palettes with elements to choose from. As mentioned before, it is important that the resulting drawings still reflect the “work-in-progress” character that is necessary to elicit appropriate feedback from users. This need has been recognized by palette creators, so that palettes with standard user interface elements are available that look like they were sketched by hand, for example the Konigi OmniGraffle Sketch Stencils [Kon12].

User interface design tools from the second category are targeted at the final design of a user interface. Therefore they usually only provide user interface elements that are available for a given platform, which restricts the creative freedom that is necessary in the beginning of a project to evaluate the design space. Also, as they are meant to be used for final software development, the resulting user interfaces necessarily look rather finished, which works against the intentions of the SCRIPT framework, and should thus be avoided.

6.5 System Specification

The SCRIPT model focuses on the description of a small but representative set of concrete interactions between user and system. As such, it does not constitute a general model of the system to be built, but may well be the basis for the development of such a model that takes the role of a system specification. While certain development methodologies, especially from the agile community, argue that such a model is not needed anyway, many established methodologies require the existence of a formal and complete system specification. Models created from the SCRIPT model can be the basis for and part of such a specification.

The analysis model is one of the first models that are created during software development and that is part of the specification. It gathers and formalizes information about the application domain and the system to be built so that it can be checked for completeness, correctness and consistency. The analysis model consists of three parts: a functional, a dynamic and an object model. The Unified Modeling Language (UML) provides various diagram types for capturing and visualizing each of these models. Most commonly used are use case diagrams for the functional model, activity, state and sequence diagrams for the dynamic model, and class diagrams for the object model. In the following sections, the relationship between the SCRIPT model and selected UML concepts are shown.

6.5.1 Deriving Use Cases

Compared to narrative scenario descriptions, the scenario meta model and interaction meta model of the SCRIPT framework have the advantage that they already provide some structuring for a scenario. `UserAction` and `SystemAction` as defined in the interaction meta model are similar to the steps in a flow of events of a use case [Coc01]. In general, the scope of a scenario is broader than that of an use case. It captures context information that might not show up in a use case as it does not directly influence the interaction between user and system, and it can span many different functionalities of a system in a sequence that may be specific for that concrete scenario. In contrast, a use case focuses on a rather small, coherent set of interactions that are required by a user in order to solve a single task. Another important difference between scenarios and use cases is that a use case abstracts from concrete data used in an interaction and that it also allows to define exceptional behavior.

The relation between the scenario meta model and interaction meta model on the one hand, and the use case meta model on the other hand can be seen in Figure 6.3. As a `UsageScenario` is meant to describe a longer thread of interactions, it can relate to multiple `UseCases` that each define a coherent subset of the interactions. In case multiple `UsageScenarios` exist for similar interactions in order to describe normal and exceptional behavior, it is also possible that a single `UseCase` relates to multiple `UsageScenarios`.

A `UsageScenario` is divided into multiple `ScenarioParts`. Whenever a `ScenarioPart` is of type `InteractionDescription`, it is related to an according `Action`. A `UseCase` typically consists of `UseCaseSteps` that describe the flow of events of an interaction. `UseCaseSteps` are only concerned with those parts of a scenario that relate to user–system interactions, so that the association is between `UseCaseStep` and `Action` rather than to `ScenarioPart`. The single actions defined in a `UsageScenario` are too detailed for a `UseCaseStep`, as they also describe the specific behavior of the corresponding scenario prototype. To achieve the abstraction necessary for a `UseCase`, it can be necessary to relate multiple `Actions` to a single `UseCaseStep`. As mentioned before, there can be multiple `UsageScenario` describing regular and exceptional behavior, which are both defined in the same `UseCase` and which share certain `UseCaseSteps`.

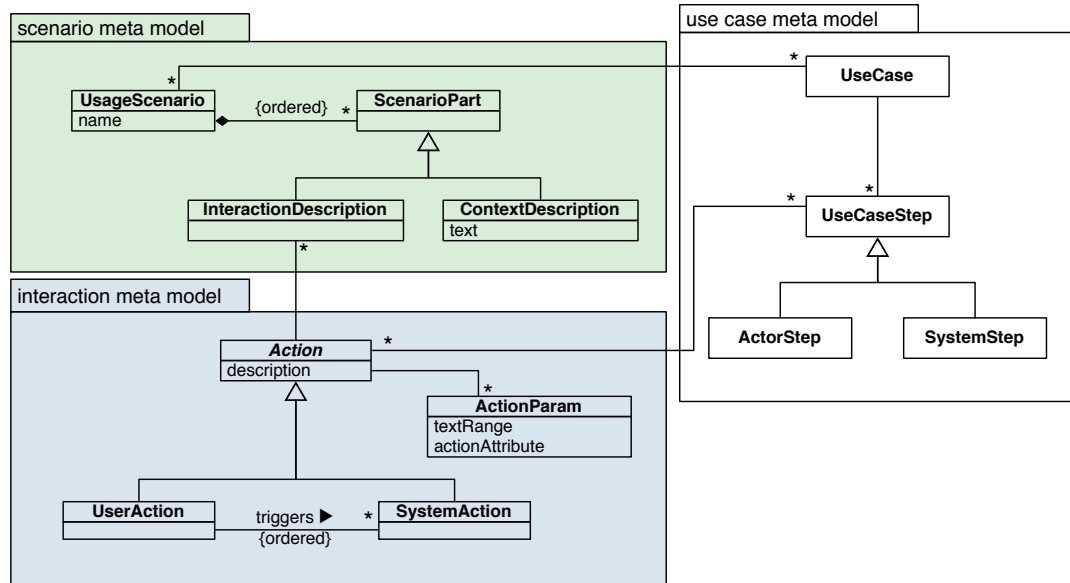


Figure 6.3: Relation between scenario meta model, interaction meta model, and use case meta model

6.5.2 Extracting User Interface Model

The scenario prototype structural meta model focuses on the visual appearance of scenes for use in a scenario prototype. As such, it not only defines which elements are visible in a scene, but also which concrete data they display, which is most often determined by their image. This is also reflected in the naming of the class “GUIElementAppearance” in the scenario prototype structural meta model. It not only represents an element of the user interface, but also its concrete state that is relevant for the GUIScene it is located in, hence its concrete “appearance”.

When it comes to define the generic layout of the user interface, it is necessary to generalize from this concrete representation in order to remove redundancies and make the resulting user interface model more precise. The resulting model can be seen in Figure 6.4. Multiple GUIScenes that are based on the same layout of the user interface and that only differ in the concrete data they show, can be abstracted to a GUIScreen. Accordingly, multiple GUIElementAppearances that represent the same logical element can be abstracted to a GUIElement. A GUIScreen then contains multiple GUIElements and a GUIElement can serve as a container for other GUIElements, analog to the relations between GUIScene and GUIElementAppearance. Additionally, a GUIScreen can be related to multiple other GUIScreens, as can be seen by the many-to-many relationship on GUIScreen. This association represents the possible ways of navigating between GUIScreens. This association also constitutes the basis for visualizing GUIScreens and the navigation paths between them in a navigation map (see Section 4.4.2).

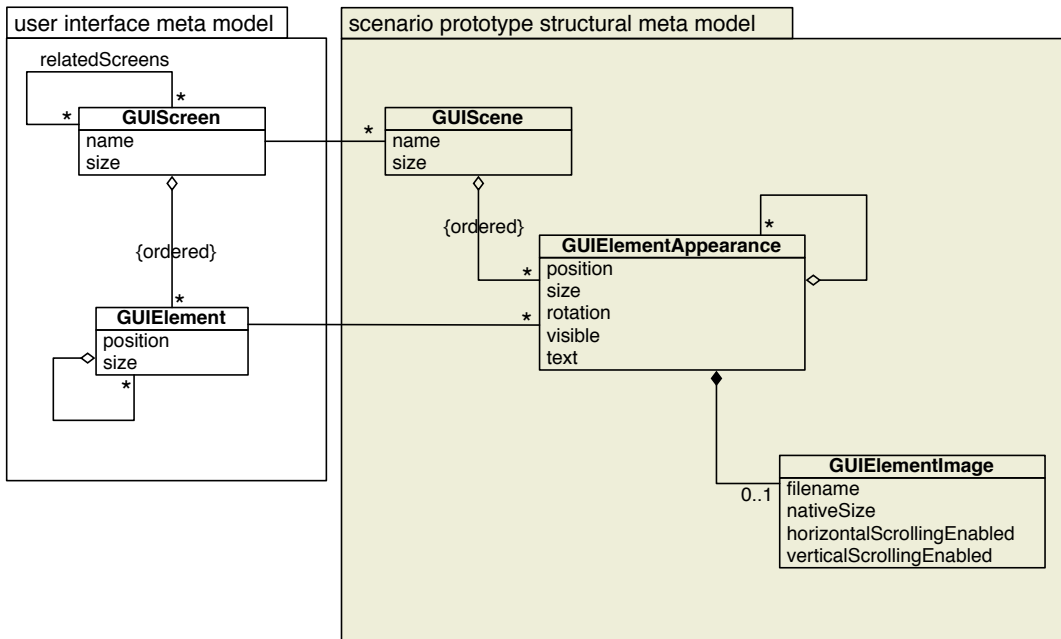


Figure 6.4: Relation between scenario prototype structural meta model and user interface meta model

The generalization of `GUIElementAppearance` to `GUIElement` is also a necessary prerequisite for the identification of analysis model elements that is described in the following section.

6.5.3 Identifying Analysis Model Elements

In this section, the focus is on the object model of the application domain and the system under development. This model is called analysis model. It is represented by class diagrams, in particular with classes, attributes, methods and associations between classes. While attributes and methods eventually need to become part of some class, it is possible that they are identified before their containing class has been found. Until then, they may exist as elements in their own right. With respect to the SCRIPT framework, both scenarios and scenario prototypes can be analyzed regarding their ability to add to the definition of the analysis model.

Although the scenario meta model does provide some structuring of the scenarios, the concrete content of a scenario is only available in natural language. The problem of extracting system specifications from natural language has been tackled by various researchers. One of the first was Abbott [Abb83, Abb87] back in 1983. Although at that time there was yet no notion of object-orientation, his technique for extracting formal system definitions from natural language descriptions has become the basis for many other approaches. Kof [Kof10] and others [KNS⁺08] who work in the field of natural

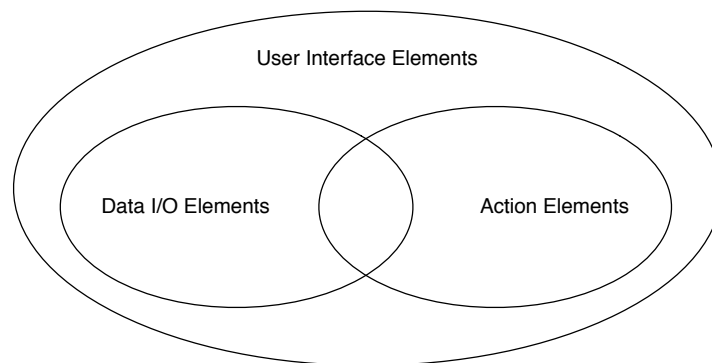


Figure 6.5: Roles of user interface elements

language processing (NLP) have further refined and automated this technique, which was a purely manual task in the beginning. These approaches can also be applied to the textual descriptions of UsageScenarios defined by the scenario meta model in order to identify elements of the structural model.

A new possibility for identifying elements of the analysis model is posed by the scenario prototype structural meta model. GUIScenes and GUIElementAppearances defined by it can be used for identifying necessary elements in the analysis model. Mrdalj and Jovanovic follow a similar approach [MJ02], but on a more abstract level. They first define a use case model, develop according prototypes and use those to derive the required elements of the analysis model. Although the approach sounds very similar to SCRIPT in general, it differs in some crucial aspects. First, they assume users of their approach to be familiar with UML, which excludes designers with no modeling background. Second, they rely on use cases, which, although they focus on the user of a system, bear the risk of misunderstandings and differing expectations due to their abstract level. And third, they provide no details about where the prototypes come from, what their structure is and how they are related to use cases. Also, no concrete description of how to identify structural model elements from prototypes is given.

Role of User Interface Elements

In order to identify elements of the analysis model from scenario prototype meta models, the role of each user interface element needs to be analyzed. The Venn diagram presented in Figure 6.5 shows the possible roles a user interface element can take. Note that it refers to single user interface elements, i.e. elements that cannot sensibly be divided into smaller parts, like labels or single images. The total set of all user interface elements is represented by the outer circle. It constitutes all visible elements of a user interface. A subset of these elements may belong to the subset of Action elements, to the subset of Data I/O elements or both. Elements from the Action subset, e.g. buttons, enable the user to trigger some operation on the underlying system. Here, the focus is only

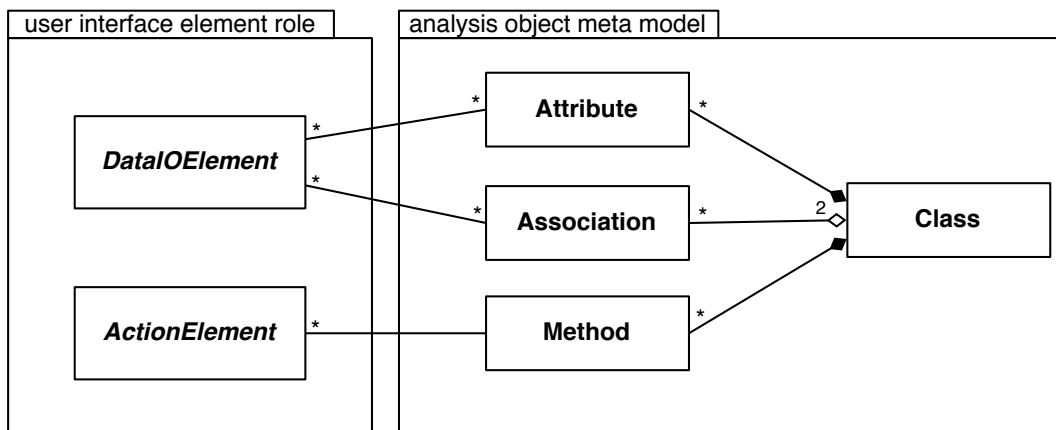


Figure 6.6: Relation between role of an user interface element and analysis meta model

on actions that invoke some business logic in the underlying system, and that do not only result in mere changes on the user interface. Elements from the Data I/O subset, like text fields and labels, are used for data input by the user or for displaying data to the user by the system. As the intersection shows, elements can be part of both Action and Data I/O subset. Elements that belong to neither subset are static user interface elements like logo images or similar.

The classification helps to decide if a user interface element relates to an attribute, an association or a method of a class in the analysis model, as visualized in Figure 6.6. The roles from the Venn diagram in Figure 6.5 are depicted as abstract classes. User interface elements that belong to the Action subset most likely correspond to a method in the analysis model, where, as before, the focus is on methods that invoke some business logic and not only perform actions on the user interface. Elements from the Data I/O subset can relate to one or more attributes and information about one or more associations, for example the amount of referring objects. The attributes are not required to be part of the same class, as the intention of the user interface element can be to display aggregate information. For user interface elements that belong to the intersection of Action and Data I/O subsets, all of the afore-mentioned applies. Like the scenario meta model and interaction meta model, this kind of mapping between user interface elements and analysis model elements is technology independent and therefore not only applicable to 2D GUIs that are the primary focus of the SCRIPT framework.

Mapping between Scenario Prototype Meta Models and Analysis Model

The elements that are readily available from a scenario prototype are instances of GUIScenes and GUIElementAppearances. However, multiple instances can represent the same concept, only with other concrete data based on the underlying scenario. As analysis model elements abstract from the concrete data, the more abstract notions of

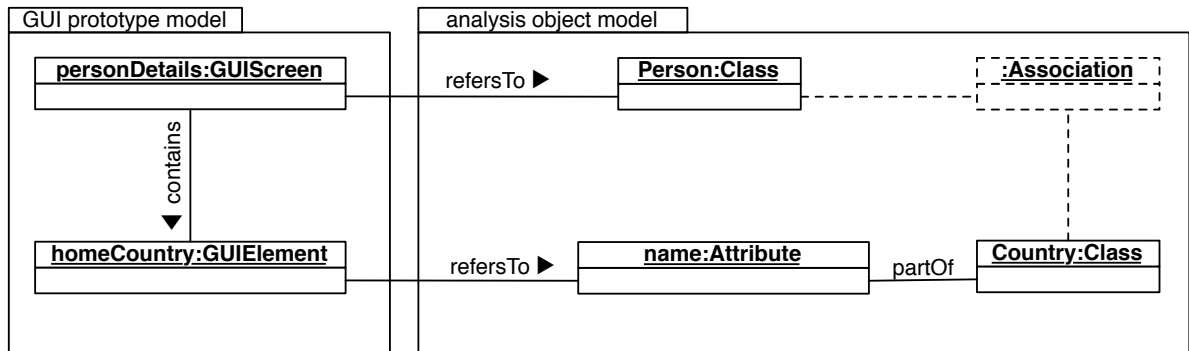


Figure 6.7: Example for identification of attributes, classes and associations

GUIScreen and GUIElement as described in Section 6.5.2 are the primary source for analysis model element identification.

First, GUIElements that contain no nested GUIElements are analyzed in more detail. They correspond to user interface elements as discussed in the previous paragraph. A single GUIElement can however contain multiple user interface elements, as this kind of granularity may have been sufficient for the creation of the scenario prototype. So in order to map a GUIElement to one or more analysis model elements, it has to be decomposed into single user interface elements. This can be done by visually inspecting the related GUIElementAppearances. Each of the contained user interface elements is then assigned one or more roles according to Figure 6.5. Depending on the roles that have been identified, the relation between role and meta model as depicted in Figure 6.6 can be used to decide if the element in questions contributes to the identification of one or more analysis model elements.

In case a GUIElement has nested GUIElements, first the above described procedure is followed for all child GUIElements. Afterwards, the procedure is applied to the containing GUIElement, taking care that already identified analysis model elements from child GUIElements are not referenced repeatedly.

GUIScreens can also contribute to the identification of analysis model elements, mostly to the identification of Classes. Especially on mobile devices, where display size is limited, it is likely that the whole GUIScreen is used to represent a single entity. In case a GUIElement that is part of this GUIScreen relates to Attributes of another Class, this also leads to the identification of Associations between the Class represented by the GUIScreen and the Classes that contain the Attributes represented by the GUIElement.

To visualize this situation, consider the example displayed in the instance diagram of Figure 6.7, which describes an excerpt from a typical address book application. In the GUI prototype model, a GUIScreen called “personDetails” exists that displays information about a person. It has a GUIElement named “homeCountry”. The GUIScreen has been analyzed and it has been found to relate to a Class called “Person” in the analysis model. The GUIElement displays the name of the home country that has been selected

for a person, hence the Attribute “name” of the Class “Country” is identified. As the GUIElement is part of the GUIScreen, this also triggers the identification of an Association between the Classes “Person” and “Country”. This is depicted by the dashed lines in Figure 6.7.

So far, only elements from the scenario prototype structural meta model have been used for identifying elements of the analysis model . But also the Actions defined in the scenario prototype interaction meta model can help in element identification. At a first glance, a GUIElementAppearance that has a UserAction defined on it might look like a good candidate for the identification of a Method in the analysis model . This has to be treated with caution though, as a UserAction primarily relates to a change on the user interface that does not necessarily have influence on the business logic. A typical example where this is the case is a UserAction that is actually only concerned with a user entering data. A more detailed description on this kind of interactions is given in Section 5.3.

Instead, a ChangeSceneAction can be a good indicator for identifying an Association between two Classes. Consider the case where a GUIElement corresponds to an Attribute or a Method of Class A, and the GUIScreen that is the target of the ChangeSceneAction corresponds to Class B. This suggests a strong relationship between those two Classes and may hence result in the identification of an Association between them.

The heuristics presented in this section can only play a supporting role in the identification of analysis model elements from scenario prototypes. Their suitability largely depends on the scenarios that have been realized and the way the scenario prototypes have been constructed. The resulting analysis model will most likely not be complete, as the available scenarios and scenario prototypes typically do not cover all aspects of the system, but only the most relevant ones. Although manual processing still takes a dominant role in this activity, the scenario prototype meta models can nevertheless provide a basis for the identification of analysis model elements.

6.6 Document Export

In addition to the direct extraction of system models, the SCRIPT models allow to export several types of documents. These can be used as a basis of discussion and for information exchange between designers and stakeholders.

6.6.1 Static Documents

The SCRIPT models allow to generate static documents, both textual and visual. An overview of possible combinations is given below:

Text only

The first way of creating static documents is the creation of traditional, narrative scenario descriptions. In order to create a textual representation of a UsageScenario, all ScenarioParts are analyzed and the extracted text fragments are appended to each other. For a ContextDescription, the text from the text field is extracted. In case of an InteractionDescription, the text of the description field of the connected Action is extracted. If the Action is of type UserAction, also the texts from the description fields of all triggered SystemActions are collected.

Images only

The SCRIPT models also enable the export of scenarios as a sequence of images that depict the state of the user interface after each interaction, without any accompanying textual description. This way, storyboards of scenarios can be created and it can be evaluated if the flow of screen states is understandable even without additional textual explanations. Depending on the desired level of granularity, it can be chosen whether a separate image, i.e. a “screenshot” should be created after each SystemAction, or only after a group consisting of a UserAction and the SystemActions it triggered, has been evaluated.

Combination of text and images

It is also possible to create documents that combine both the expressive power of images and the textual explanations that compensate for ambiguities that might arise from the visual representations alone. As in the “Images only” case above, the desired level of granularity for image creation can be chosen. In case only an image of the visual appearance before and after a group of UserAction and triggered SystemActions is exported, the intermediate steps can be deduced from the accompanying textual description. An example for such a type of export can be seen in Appendix A, where static exports of the prototypes that have been used for the evaluation of the SCRIPT framework are presented.

6.6.2 Video Generation

The SCRIPT framework is not restricted to the generation of static documents. As scenarios are not only available in textual, unstructured format, but also have an underlying model describing the flow of interactions in detail, it is possible to exploit this information in order to create animated videos of the described interactions. Chang and Ungar realized the importance of providing the user with visual clues of what is happening on the user interface. They promote to use animations whenever possible for user interface changes such as appearing, disappearing or moving windows [CU93].

While their argumentation is geared towards development of user interface frameworks, it is also applicable in the context of documenting the flow of interactions in a scenario.

Instead of creating a static document as described in the previous section, the SCRIPT model can be used for creating a video of the flow of interactions of a scenario. To achieve this, the ScenarioParts of a UsageScenario are analyzed sequentially. If the ScenarioPart is of type ContextDescription, it does not describe any interaction between user and system, but rather about the context of usage. In the resulting video, the textual description defined by it can be displayed as a text panel or a synthesized voice-over narration can be generated using text-to-speech technology.

Each InteractionDescription can be directly visualized by executing the UserActions and SystemActions that it relates to. The resulting video is then similar to a screen recording of a user interacting with the prototype, only that it does not require this work to be done manually. Another advantage of automatic video creation over manually creating screen recordings is that changes in the prototype can be directly reflected in videos by simply regenerating the video. The tedious task of recreating a screen recording every time a prototype is modified—with all potential post-processing work that can be involved—is no longer necessary.

Chapter 7

The SCRIPT Editor

In order to evaluate the SCRIPT framework, a prototypical tool—the SCRIPT Editor—was implemented that allows to simultaneously edit prototypes and textual scenarios.

The current implementation allows to create prototypes for devices running on Apple iOS. As explained above, the SCRIPT framework is applicable for all kinds of development projects that require a user interface, be it smart phones, desktop applications, or even embedded systems, as long as they provide a graphical user interface. The SCRIPT Editor features a one-click integration with the iOS-Simulator, so that prototypes can be instantly tested while editing. It is also possible to deploy a prototype on a device running iOS and thus evaluate it in its designated context of use.

The chapter is organized as follows: Section 7.1 presents the user interface of the SCRIPT Editor and explains how model inconsistencies are immediately communicated to the user. Section 7.2 gives an overview of the architecture of the SCRIPT Editor and the frameworks that are involved in its implementation. Section 7.3 presents the components that have been used in order to set up a workflow for creating and deploying prototypes for iOS devices.

7.1 User Interface

The user interface of the SCRIPT Editor is split into three parts, as can be seen in Figure 7.1. On the left side is the Navigator, which displays the GUIScenes and UsageScenarios that are contained in a project. GUIScenes have the icon of a small camera and UsageScenarios that of a text page. For organizational purposes, the concept of a folder has been added so that it is possible to give the project some hierarchical structure. Folders can be nested inside each other and can contain both GUIScenes and UsageScenarios.

The large gray area on the upper right side is reserved for the Scenario Prototype Editor and Scenario Editor, which will be explained in detail below.

On the lower right side, tabs for multiple views are located. The Properties view is used for showing properties of a selected element if available. The Validation view shows information about validation errors that occur whenever scenario prototypes and scenarios become inconsistent. An example for this situation is given below. The Emf-

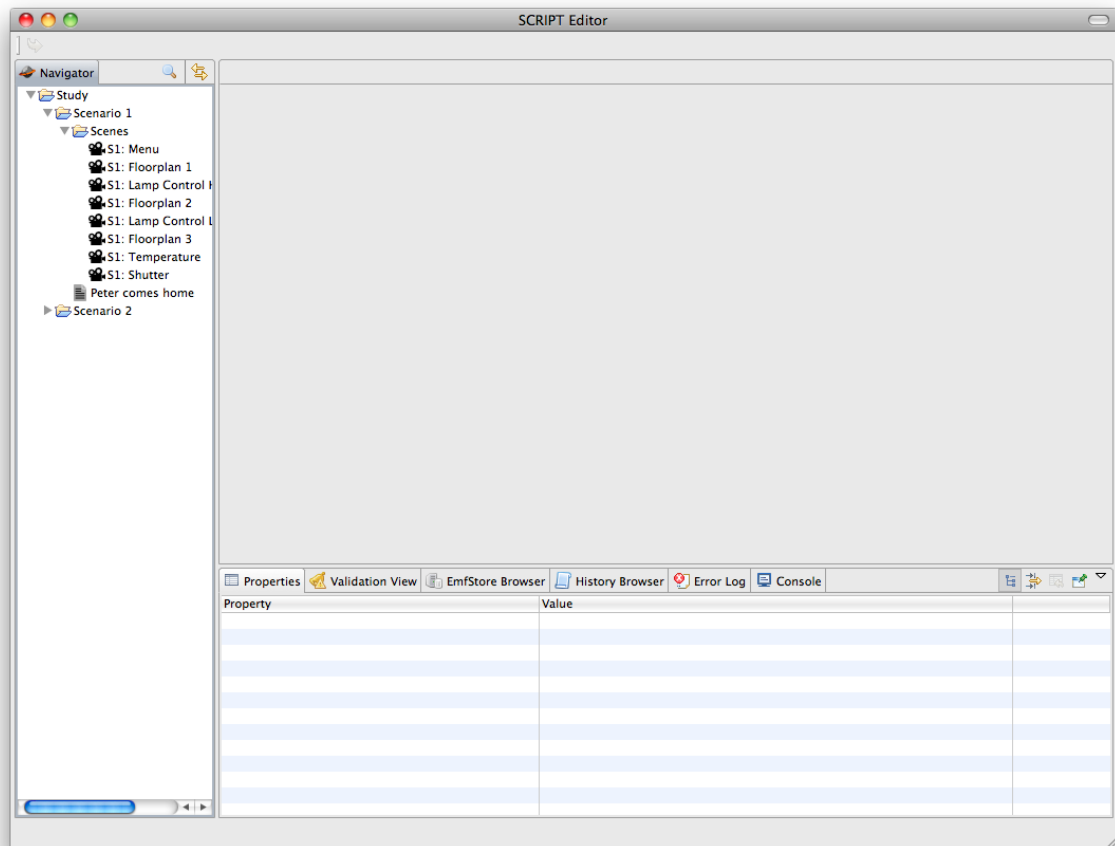


Figure 7.1: The SCRIPT Editor

Store Browser allows to define and access remote repository locations. Projects can be published to these repositories so that they can be retrieved by other users, and changes can be committed and retrieved from a repository. The History Browser can be used to track the changes that have been made to a model element. Finally, the Error Log and Console are used for debugging purposes.

In the next two sections, the Scenario Prototype Editor and the Scenario Editor are presented in detail. Note that the SCRIPT Editor itself is still in a prototypical state and currently requires to first create a prototype and then edit the related scenario. This corresponds to path (b) of model traversal as explained in Section 6.3.

7.1.1 Scenario Prototype Editor

The user interface of the Scenario Prototype Editor is shown in Figure 7.2. It can be opened up by double-clicking a GUIScene in the Navigator on the left. The editing

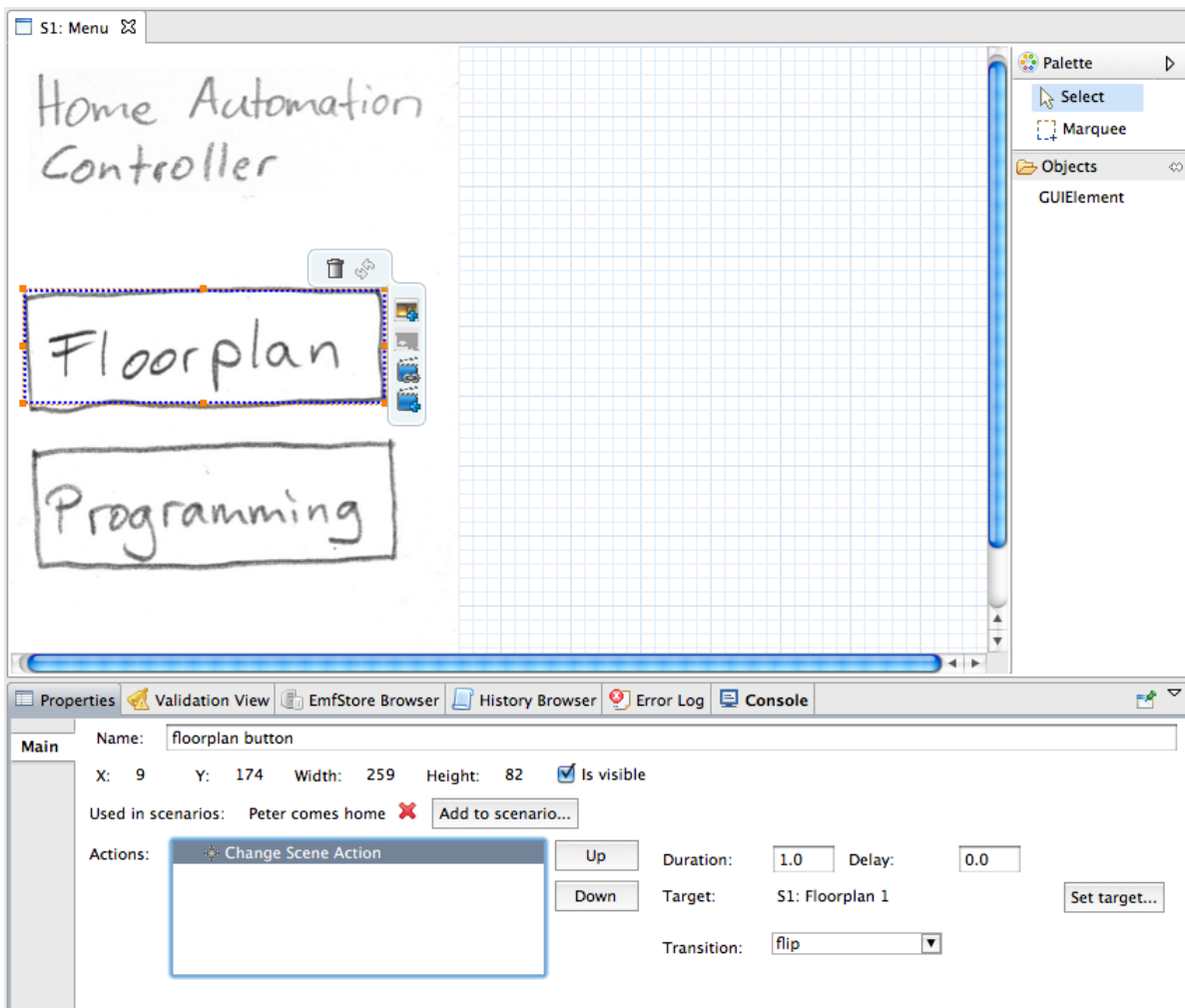


Figure 7.2: The Scenario Prototype Editor

area has a light blue, checkered background, which makes it easier to distinguish it from images with white background. In Figure 7.2, a `GUIElementAppearance` with the scanned image of a hand-drawn menu (“Home Automation Controller”) has been defined as background. On top of that, a transparent `GUIElementAppearance` is located on the area of the pictorial “Floorplan” button, visualized by the dashed blue border. `GUIElementAppearances` can be created using the “`GUIElement`” entry on the palette on the right side of the editor.

As the transparent `GUIElementAppearance` is currently selected, a popup menu is shown for performing actions on the `GUIElementAppearance`. Apart from deleting and updating the `GUIElementAppearance`, an image can be set or removed, and actions can be defined that should be triggered once a user clicks or taps on the `GUIElementAppearance`.

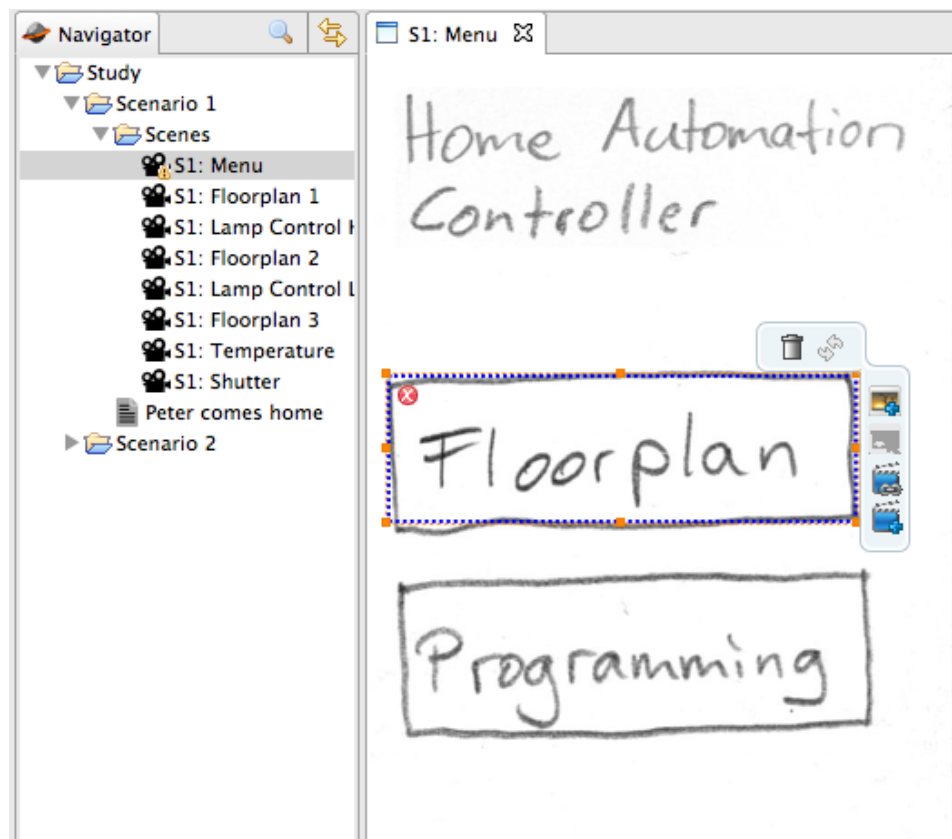


Figure 7.3: The Scenario Prototype Editor, showing an inconsistency between scenario prototype and scenario, which is displayed with an error icon in the Scenario Prototype Editor and a warning icon in the Navigator

The Properties view on the lower right shows various attributes relating to a `GUIElementAppearance`. To ease the identification of `GUIElementAppearance`s, a name can be defined. In the next row, the coordinates of the `GUIElementAppearance` are displayed, and it can be selected whether the `GUIElementAppearance` should be visible by default. This can be deselected if it should only become visible in response to an user interaction with another `GUIElementAppearance` of the `GUIScene`.

In the next row, it can be defined to which `UsageScenarios` the interaction with this `GUIElementAppearance` belongs. This corresponds to the association between the class `InteractionDescription` from the scenario meta model and `Action` of the interaction meta model. Whenever the `Actions` of a `GUIElementAppearance` are added to a `UsageScenario`, the `SCRIPT Editor` automatically creates a new `InteractionDescription` and adds it to the list of `ScenarioPart` of the `UsageScenario`.

The last row is used for defining the actions that should be triggered by clicking (or tapping) a `GUIElementAppearance`. The editor currently only supports the definition of `UserClickActions`, hence it is not mentioned explicitly, but only the resulting `System-`

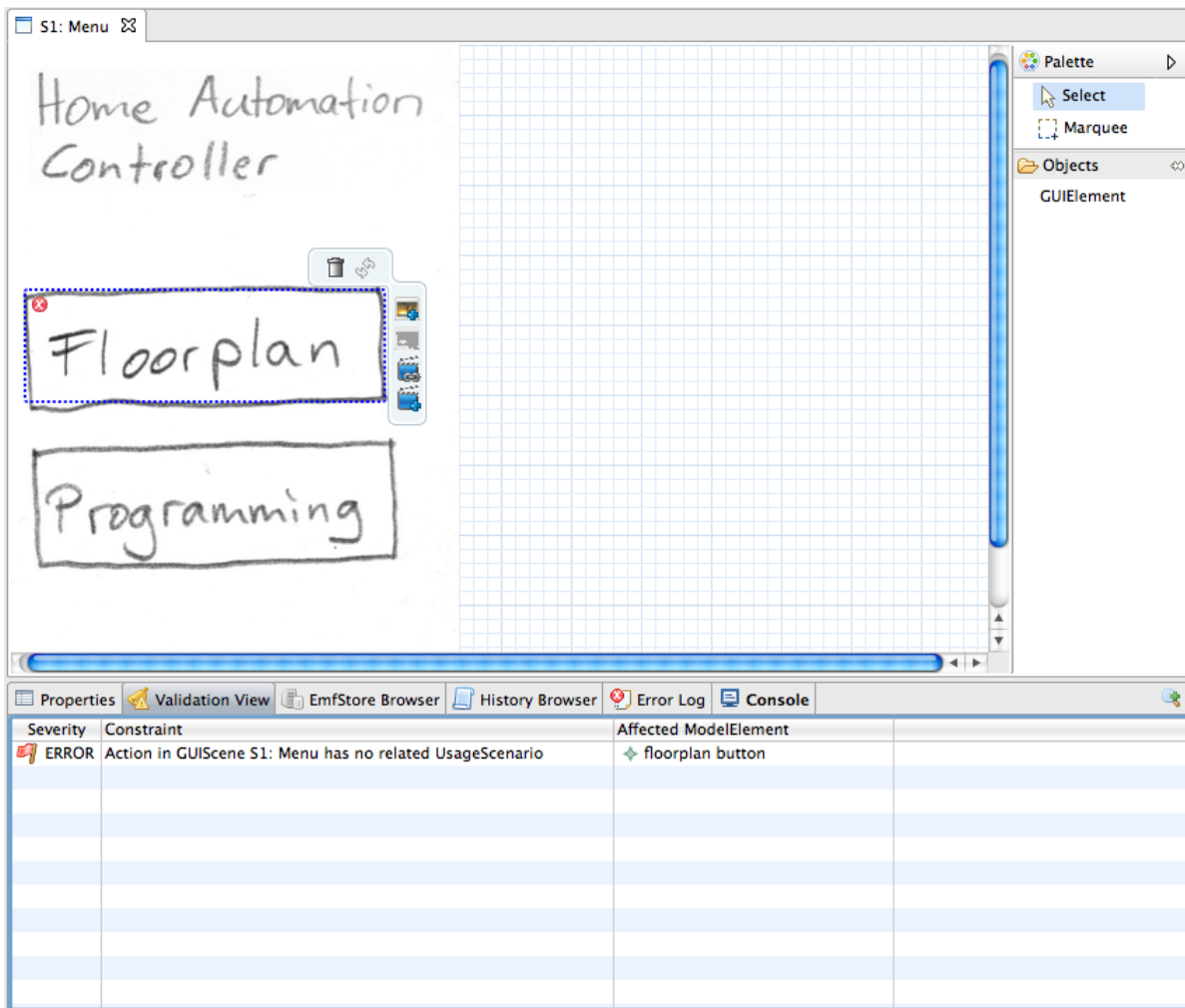


Figure 7.4: The Validation view, listing an inconsistency between scenario prototype and scenario, which is shown as a validation error

Actions are listed. Whenever the first triggered SystemAction is defined, the triggering UserClickAction is created automatically, and removed upon removal of the last remaining SystemAction. The SystemActions are executed sequentially as they are displayed in in the list. Multiple SystemActions can be grouped in a ParallelActionGroup. When a SystemAction is selected, its properties are shown on the right of the list. In the case of the ChangeSceneAction shown in Figure 7.2, duration and delay are set to their default values. The field “target” defines which GUIScene should be shown next, and the field “transition” allows to define an effect that should be used for switching GUIScenes.

Whenever there is an action defined for a GUIElementAppearance, there also needs to be at least one UsageScenario to which it is related. If this is not the case, there is an inconsistency between scenario and scenario prototype, which is communicated to the user

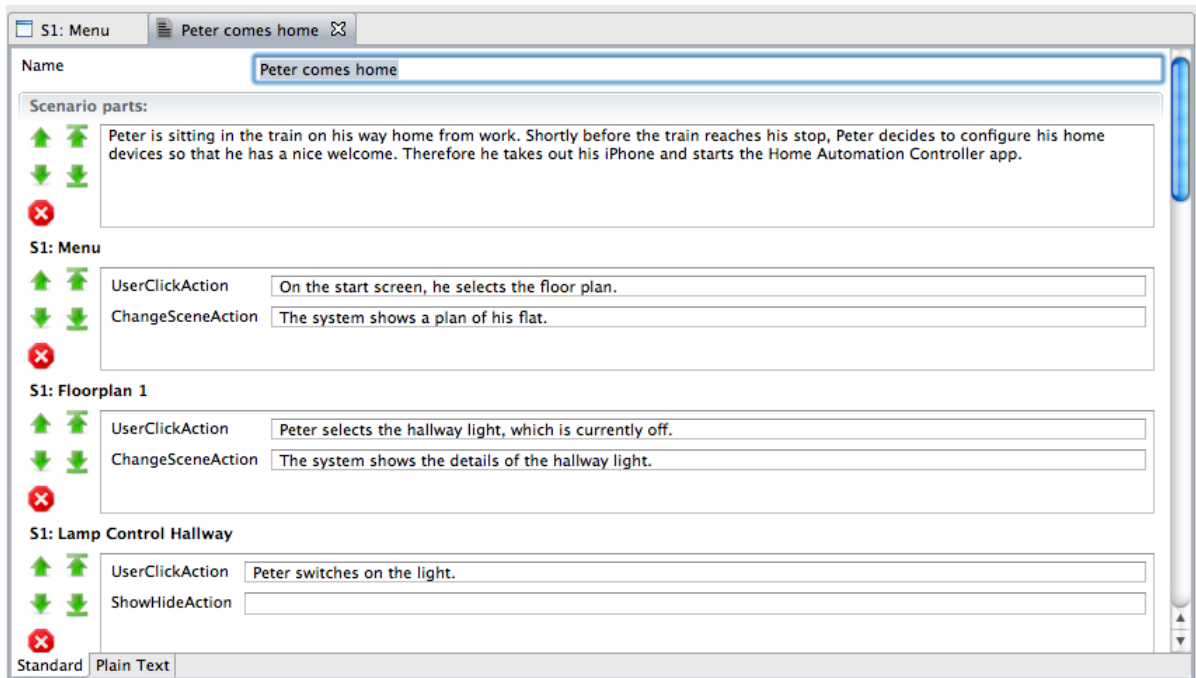


Figure 7.5: The Scenario Editor

as an error. Consider the situation in Figure 7.3, which is similar to Figure 7.2 but no UsageScenario has been related to the actions defined for the GUIElementAppearance. The user is informed about this inconsistency with an error icon that is displayed on the affected GUIElementAppearance. A warning icon is added to its containing GUIScene in the Navigator, so that the error does not get overlooked when the respective Scenario Prototype Editor is not open. Figure 7.4 shows the according error entry in the Validation view.

7.1.2 Scenario Editor

The user interface of the Scenario Editor is shown in Figure 7.5. Apart from the name of the UsageScenario, which can be defined in the top-most text field, the majority of the editor is devoted to the ScenarioParts of the UsageScenario. Each ScenarioPart has a set of control at its front, which allows to move it up or down one position, move it to the first or last position, and delete it from the UsageScenario.

A ScenarioPart can be of type ContextDescription or InteractionDescription. If it is a ContextDescription, it should describe the context of system usage, with no user–system interactions taking place. To enter this information, the Scenario Editor provides a large text field, as can be seen by the first entry in Figure 7.5.

In case a ScenarioPart is of type InteractionDescription, then for the related Action a text field is displayed that corresponds to its description field. In case of a UserAction, a

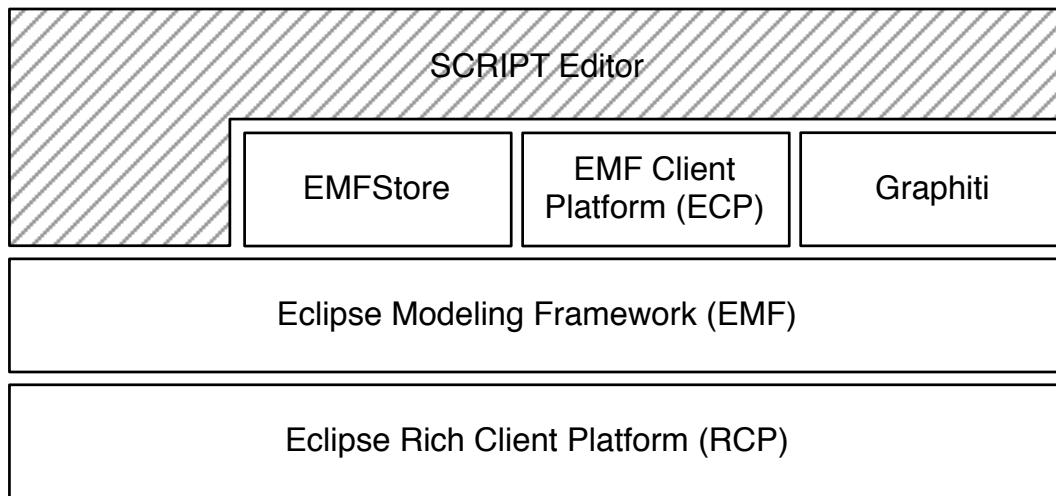


Figure 7.6: Layers of the SCRIPT Editor

text field is shown for each SystemAction that is triggered by the UserAction. Whenever the GUIScene that is shown to the user changes, this is indicated by the name of the newly shown GUIScene being printed in bold letters.

The behavior of deleting a ScenarioPart is slightly different depending on the type of the ScenarioPart that is being deleted. When the ScenarioPart is of type ContextDescription, it is simply deleted from the model, and hence the textual description stored with it needs to be re-entered should it be required later on. When the ScenarioPart is of type InteractionDescription, it is deleted from the model as well. In this case, however, the textual descriptions are preserved, as they are part of the related actions, which belong to the scenario prototype and hence do not get deleted. If the actions defined in the scenario prototype are not related to any other UsageScenario, the removal of the InteractionDescription triggers a validation error and the user is informed about the inconsistency.

7.2 Architecture

The software architecture of the SCRIPT Editor is a layered architecture as depicted in Figure 7.6. The lowest layer is the Eclipse Rich Client Platform (RCP) [Ecl12e]. Eclipse RCP allows to derive native GUI applications for various operating systems, such as Windows, Linux and Mac OS X from the same source.

The second layer is the Eclipse Modeling Framework (EMF) [Ecl12a]. EMF provides facilities for data modeling and handling, including automatic code generation for model classes, adapters and editors, automatic change tracking and notification, and data serialization. EMF also provides a validation framework that allows to define constraints on a model which need to hold in order for a model to be considered valid. Whenever a

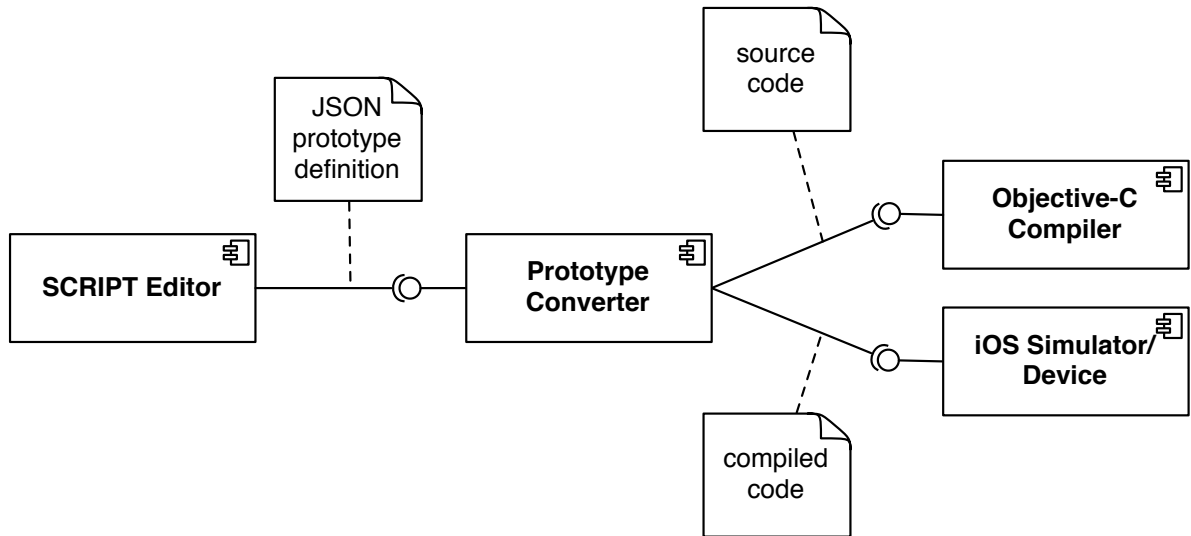


Figure 7.7: Components of the SCRIPT evaluation setup

constraint does not hold, the validation framework throws an error, which can be handled automatically by application logic, or displayed to the user. In the SCRIPT Editor, the validation framework is used to ensure consistency between scenarios and scenario prototypes. The SCRIPT model has been defined using EMF and the according model classes have been generated from the model.

The third layer consists of EMFStore, the EMF Client Platform (ECP) and Graphiti. EMFStore [Ecl12c] is a version control system similar to Subversion, but instead of files in the filesystem it operates on models that are based on EMF. EMFStore tracks all changes that are made to models and provides facilities to send changes to a common repository and retrieve and apply changes of other users respectively.

ECP [Ecl12b] provides generic browsing and editing functionalities for EMF-based models. The navigator view of ECP visualizes models in a tree-like structure, and the generic editor of ECP provides a user interface to inspect and modify attributes of model elements via reflection at runtime, without any manual coding necessary.

Graphiti [Ecl12d] is a graphics framework that integrates with EMF-based models and that provides a sensible set of default functionality and behavior, so that implementation efforts can be concentrated on the business logic. In the SCRIPT Editor, Graphiti was used for creating the Scenario Prototype Editor.

7.3 Components of Evaluation Setup

For the evaluation of the SCRIPT framework, a set of components has been realized to construct a workflow that allows to create prototypes for the Apple iOS platform. Parts of it, however, are platform independent, so that workflows for other platforms can be

created easily. Figure 7.7 shows the components that are involved in the workflow and the types of artifacts that are exchanged.

Just like the SCRIPT framework in general, the SCRIPT Editor is platform agnostic and allows to create prototypes for a wide range of platforms, no matter if for mobile, desktop, or even embedded computing. The prototypes that are created using the SCRIPT Editor can be exported in JSON (JavaScript Object Notation) format together with all used image files for exchange with other editors, for compilation, or for interpretation. The advantage of the JSON format is its compact notation, which reduces file sizes e.g. for transmission over the Internet, while still being human-readable.

In the course of this dissertation, the goal was to create prototypes that can be executed on a device running Apple iOS. This could have been achieved by either creating a generic playback engine that directly reads the JSON prototype definition and interprets it on the iOS device, or by creating an intermediate tool that creates a standalone native iOS application, which can then be deployed like any regular application. As the Apple policies for iOS application development prohibit the creation of applications that can load and execute additional code during runtime, the latter approach was chosen.

The Prototype Converter reads the prototype specification in JSON format and first creates a set of Objective-C classes from it. This has the advantage that for very sophisticated prototypes, it is possible to intercept this step and add custom code to the prototype. The source code is then compiled using the standard Objective-C compiler, and the resulting application is started in the iOS Simulator or can be deployed on an iOS device for testing in its desired use context.

Chapter 8

Evaluation

One of the main goals of the SCRIPT framework is to keep scenario prototypes and scenarios consistent during development. In order to assess the effectiveness of the SCRIPT framework with respect to this goal, a controlled experiment was conducted, which is described in this chapter.

Section 8.1 describes the design of the experiment and the hypothesis that underlies it. Section 8.2 gives an overview of the setup of the experiment and explains the tasks the participants were presented with. Section 8.3 discusses the results and findings from the evaluation.

8.1 Experimental Design

Keeping artifacts consistent means that for every change to one artifact, it has to be determined if also changes to other artifacts are necessary, and to apply these additional changes. As an example, a change to a contextual description of a scenario does not require a change to a related scenario prototype in case the change of context does not manifest in the scenario prototype. Analogously, a minor rearrangement of elements in a scenario prototype does not require changing the related scenario. In contrast, a change to the interactions that are realized in a scenario prototype does require a change to the corresponding scenario, as otherwise the scenario and the scenario prototype would no longer describe the same flow of interactions. If changes are not properly propagated to other artifacts, they become inconsistent.

In order to determine the effectiveness of the SCRIPT framework regarding the consistency between scenarios and scenario prototypes, a quantification is needed that allows for an evaluation. Therefore, the types of changes that are necessary in order to keep scenarios and scenario prototypes consistent have been analyzed and according error types have been defined.

The resulting error types refer to a propagation of changes from a prototype to its related scenario. Although the participants of the experiment were free to decide whether they wanted to edit the scenario prototypes or scenarios first, all of them started to edit the scenario prototypes and adjusted the related scenarios afterwards.

The following four types of errors have been identified:

E1: A scenario has not been updated at all.

E2: An action that was added to the prototype has not been added to the scenario.

E3: An action that was deleted from the prototype has not been deleted from the scenario.

E4: An action has been put in the wrong position in the scenario.

The error E4 requires a little explanation. The SCRIPT framework allows to automatically check for consistency between scenarios and prototypes, meaning that every interaction described in a scenario has also been realized in a prototype, and vice versa. However, only a scenario defines a chronology on the interactions via the order of its ScenarioParts, while a prototype only defines that an interaction exists, which can be triggered by a user. Whenever a new interaction is defined on a prototype, the SCRIPT Editor requires the designer to also add the interaction to the related scenario in order to keep it consistent with the prototype. As the new interaction was not part of the scenario before, its correct position in the chronology is unknown, and hence it is appended to the end of the scenario by default. The designer then needs to bring the new interaction to the correct position in the chronology in the scenario, otherwise this is regarded as an error of category E4.

While the errors E2–E4 refer to single actions, E1 refers to a set of actions that should have been edited as part of a work package. Given these types of errors, the hypothesis underlying the experiment was formulated as follows:

H_0 : The sum of errors that are made using the SCRIPT framework is equal or higher than using separate tools for the parallel development of prototypes and scenarios.

Accordingly, the alternative hypothesis reads:

H_A : The sum of errors that are made using the SCRIPT framework is less than using separate tools for the parallel development of prototypes and scenarios.

The **dependent variable** of this experiment was the sum of errors each participant made, regardless of the category the errors belonged to. This is a rather conservative calculation, as the severity of the classes of errors can be argued to be different. Whenever someone forgot to edit a scenario altogether (E1), they could no longer make any error that belongs to one of the other categories (E2–E4). As each work package required to add or remove several actions in a scenario, an error from category E1 might thus mask several errors from the other categories, especially E2 and E3. While it would be legitimate to substitute each error of category E1 with the maximum possible amount of errors of the categories E2 and E3, for the sake of transparency all error categories were treated as equal.

The **independent variable** of this experiment was the tool setup the participants were provided with for solving the given work packages. It had two values, *unified* and

separate. The *unified* tool setup consisted of the SCRIPT Editor for both editing scenario prototypes and scenarios. The *separate* tool setup used only the prototyping features of the SCRIPT Editor, while the textual scenarios was edited using Microsoft Word, but without requiring any tool knowledge beyond simple plain text editing.

To account for differences in the experience of the participants, their level of education was chosen as **controlled variable** and a randomized block design was applied so that the participants were evenly distributed across the unified and separate tool setups. When registering for the experiment, the participants had to select their level of education, where they could choose between Bachelor student, Master student, Ph.D. student and industry professional.

8.2 Tasks

The experiment required the participants to work on scenarios and scenario prototypes for a fictive development project that had the goal of developing a mobile user interface for a home automation system. The domain of home automation was chosen as it was expected to be easily understandable by all participants, while at the same time being unusual enough to prevent a distortion of the results due to varying prior knowledge in the field. The scenarios and scenario prototypes that were given to the participants can be seen in Appendix A.

The participants were split into a *unified group* and a *separate group* according to the tool setup they had to use. Both groups were given the same initial set of scenarios scenario prototypes and had to alter them in the course of the experiment. The experiment consisted of four work packages, where each work package consisted of several tasks. A reprint of the work packages that were handed to the participants is given in Appendix B.

The experiment did not take place in parallel with all participants at once, but individual meetings with each participant were scheduled according to their preferences. During the meeting, a facilitator gave a short introduction to the study and played the role of the client for the reviews. There was no maximum working time that restricted the duration of each experiment session, but instead the fastest three participants of each group were promised a shopping coupon as a reward. This competition imposed a time pressure on the participants and made it thus more similar to a real industry project. As the experiment should resemble a project that follows an iterative development process, the participants first received only two of the four work packages. After successful completion of the first two work packages, they were handed the last two work packages.

Each experiment session went as follows:

1. First, the participant received a one-page introductory document, a training video which explained the relevant features of the SCRIPT Editor and the two initial scenarios and scenario prototypes. They had a fixed amount of time for reading

the introductory document, watch the video and get to know the scenarios and scenario prototypes. This time did not count against their competition time.

2. Afterwards, they were handed the first two work package with tasks that required them to alter the scenarios and scenario prototypes. The timekeeping for the competition started with the handout of the work packages.
3. As soon as the participant felt confident that they had solved all tasks, they notified the facilitator.
4. The facilitator then reviewed the scenario prototypes to ensure that all tasks had been solved completely and correctly. If necessary, the facilitator requested the participant to rework parts of the work packages.
5. If necessary, the participant fixed the parts of the work packages that needed rework.
6. The facilitator reviewed the changes and then handed out work packages three and four.
7. The participant worked on the new work packages and again notified the facilitator once they were finished.
8. As before, the facilitator reviewed the scenario prototypes and requested changes in case not all tasks had been solved as expected, which were then fixed by the participant.

In the introductory document, the participants were informed about the course of the experiment, including that the scenario prototypes would be reviewed during the session. However, the introductory document emphasized that they also need to keep the scenarios consistent with the scenario prototypes. To prevent that they unintentionally or deliberately omitted the editing of scenarios in order to save time, the introductory document explicitly noted that errors in the scenarios would give them a time penalty in the final evaluation of the results.

The training video presented the SCRIPT Editor and explained all functionalities that were relevant for solving the tasks. The participants had access to the video during the whole course of the experiment and were encouraged to refer to it whenever they were unclear about how to use certain functionalities. The video for the separate group had a duration of about 5.5 minutes and for the unified group of about 12 minutes. The difference in duration comes from the fact that the video for the unified group explained the Scenario Prototype Editor (see Section 7.1.1) and the Scenario Editor (see Section 7.1.2). The video for the separate group only explained the Scenario Prototype Editor. For the editing of the textual scenarios using Microsoft Word, they did not get an introduction, as they used only basic text editing features of Microsoft Word and could be assumed to be familiar with it.

	Bachelor students	Master students	Ph.D. students	Professionals	total
unified group	3	4	3	3	13
separate group	3	5	2	3	13

Table 8.1: Distribution of experiment participants to groups

	Sum of errors E1	Sum of errors E2	Sum of errors E3	Sum of errors E4	total
unified group	0	0	0	9	9
separate group	13	5	7	1	26

Table 8.2: The sums of errors that were made by the participants of each group, categorized by type of error as described in Section 8.1

8.3 Experiment Results

In total, 26 participants took part in the experiment, both students and professionals, who were distributed between the separate and the unified group. To account for differences in the experience of the participants, a randomized block design regarding their level of education was used, so that Bachelor students (6 total), Master students (9 total), Ph.D. students (5 total) and professionals (6 total) were evenly, randomly distributed between the two groups. The detailed distribution of experiment participants to groups is shown in Table 8.1.

8.3.1 Number of Errors

As described in Section 8.1, the sum of all errors in all categories for each participant was taken as measure.

The null hypothesis as described in Section 8.1 was that designers using the SCRIPT framework make the same amount or even more errors than those who use separate tools for editing scenario prototypes and scenarios. Table 8.2 shows the sums of errors that the participants of each groups made, categorized by the type of error as described in Section 8.1. To evaluate the results, the Mann-Whitney U test was chosen as the results could not be assumed to be normally distributed and the sample size was relatively small ($n = 26$). The medians of the unified and separate group were 0 and 1 errors. The distribution of sum of errors of the two groups could be shown to differ significantly on a .05 significance level and with medium effect size according to Pearson's r ($U = 45$, $n_1 = n_2 = 13$, $Z = -2.14$, $p = 0.016$, $r = 0.42$).

Hence, the null hypothesis H_0 can be rejected and instead the alternative hypothesis H_A can be accepted.

Apart from the statistical evaluation, it is worth mentioning that no participant from the unified group made any error of category E1–E3, and only some of them made errors of category E4, which means that they put actions in the wrong position in the scenario. However, all participants of the unified group complained—with good reason—about the poor usability of the Scenario Editor and the massive cognitive load it creates in its current implementation. This is because due to time constraints, only a rudimentary version of the Scenario Editor could be realized that automatically appends new actions to the end of a scenario, and that requires the user to move the action to the right position using up and down buttons. Whenever there are so many actions in a scenario that it requires the user to scroll to see all actions, not losing orientation while moving actions becomes rather challenging. Improvements on the Scenario Editor are thus likely to further reduce the number of errors that are made when using the SCRIPT framework.

8.3.2 Working Time

Due to the fact that the SCRIPT Editor was still in a prototypical state, it was assumed that the unified group needs more time than the separate group for solving all tasks of the work packages. This assumption was based on the fact that participants from the unified group not only had to learn about the Scenario Prototype Editor, but also about the Scenario Editor, including its usability flaws described in the previous section.

However, the evaluation of the working time showed that the average working time of the unified group was even slightly lower than that of the separate group, if only about one minute (1:05 hours versus 1:06 hours). The standard deviation in both groups was rather high, with about 16.5 minutes in the unified group and about 24 minutes in the separate group.

The slowest participants from the unified group and separate group needed 1 hour 43 minutes and 1 hour 50 minutes, respectively. The fastest participant from the unified group needed about 43 minutes, compared to about 39 minutes for the fastest participant from the separate group. As the difference between the fastest participants is only 4 minutes, an improvement of the Scenario Editor that results in an increased performance of as little as 10% would suffice to bring them on a par.

8.3.3 Exit Interview

After the participants finished all work packages, they were posed a set of concluding statements that they should rate on a 5-point scale between -2 and +2, where -2 stands for “totally disagree with the statement” and +2 stands for “totally agree with the statement”. The statements were as follows:

- S1: “The scenarios help me in understanding the scenario prototypes in the preparation phase.”
- S2: “I like the idea of using both scenarios and scenario prototypes.”

- S3: “The SCRIPT Editor is usable.”
- S4: “It was easy to keep scenarios and scenario prototypes consistent.”
- S5: “The error and warning icons motivated me to keep scenarios and scenario prototypes consistent.”

The last statement was only posed to participants of the unified group, as only their tool setup allowed for automatic checks of consistency between scenario prototypes and scenarios.

For S1, less participants of the unified group rated the scenarios helpful compared to participants of the separate group ($average_u = -0.08$, $average_s = 0.77$). Some of them explained this with the missing possibility to view scenario texts directly in the Scenario Prototype Editor. The responses to S2 showed that both groups were about equally positive about the combined use of scenarios and scenario prototypes ($average_u = 1.69$, $average_s = 1.62$). In the responses to S3, there is a clear shift towards the negative end in the unified group that used the Scenario Editor, which clearly indicates that it needs to be improved concerning its usability and which has also been confirmed by the participants ($average_u = 0.15$, $average_s = 1.08$). However, the tool combination seemed to reduce the subjective difficulty of keeping scenarios and scenario prototypes consistent, as can be seen from the responses to S4 ($average_u = 0.15$, $average_s = -0.15$). Finally, the majority of the participants in the unified group considered the visualization of inconsistencies between scenarios and scenario prototypes via error and warning icons as a good motivation to make them consistent again, as can be seen in the responses to S5 ($average_u = 1.62$).

On a side note, 23% of the participants in the separate group (3 of 13), mentioned they wished they had been given a single tool for both editing scenario prototypes and scenarios instead of having to switch between two tools, without knowing that this was the setup of the unified group, which used the SCRIPT Editor.

8.3.4 Threats to Validity

This section discusses how certain threats to validity of the experiment have been addressed.

In order to ensure that differences in the results of both groups could be correctly attributed to differences in the tool setup, some precautions have been taken. To cope

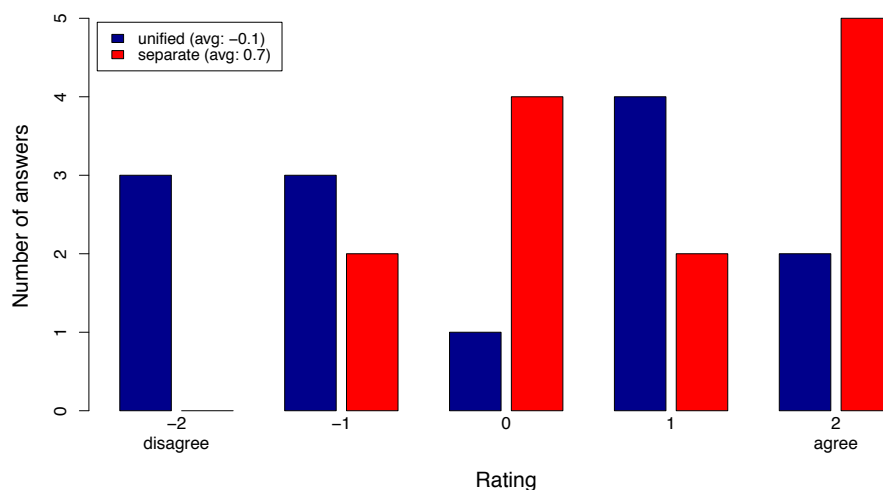


Figure 8.1: Responses to S1: “The scenarios help me in understanding the scenario prototypes in the preparation phase.”

with influences from the level of knowledge of participants, a randomized block design was used. This ensured that each group had about the same amount of people with the same educational background. The randomized assignment of people to groups prevented selection bias. To limit the influence of the facilitator as far as possible, the only personal interaction that took place was for the evaluation of the scenario prototypes during the experiment. The necessary information about the tool was provided in form of a video, and the introductory information and work package information was handed to the participants as PDF documents.

The results of the experiment can be generalized with caution. Although many participants were either students or working in a university context, 26% of the participants (6 of 26) had a professional background. The total number of participants, however, was relatively small. The experiment setup and the work packages that were given to the participants were based on experiences from real-world projects and the experiment took place in an office environment which was similar to a typical office setting.

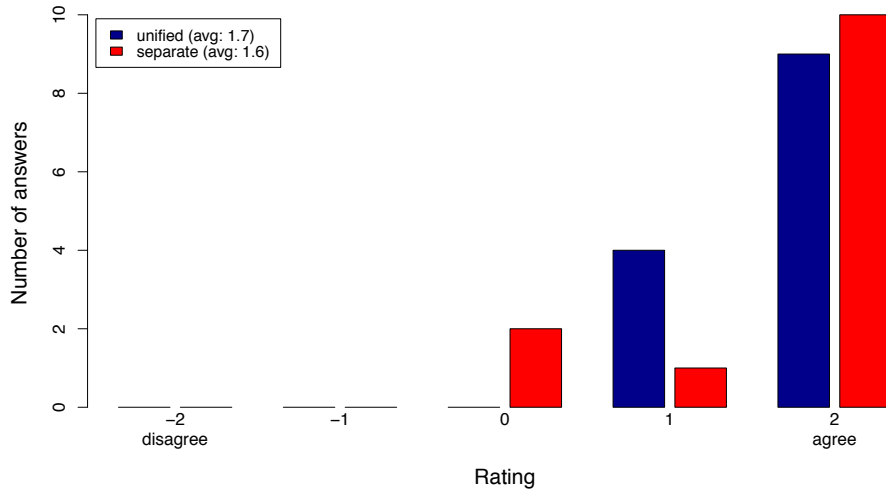


Figure 8.2: Responses to S2: “I like the idea of using both scenarios and scenario prototypes.”

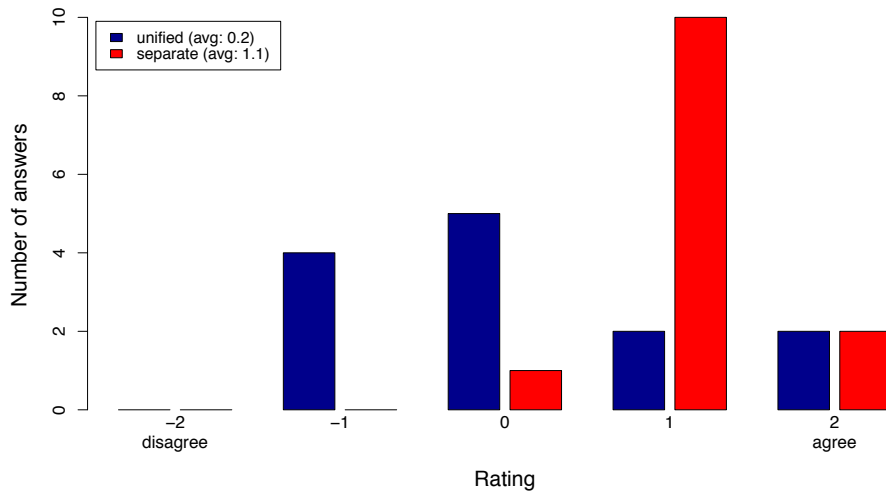


Figure 8.3: Responses to S3: “The SCRIPT Editor is usable.”

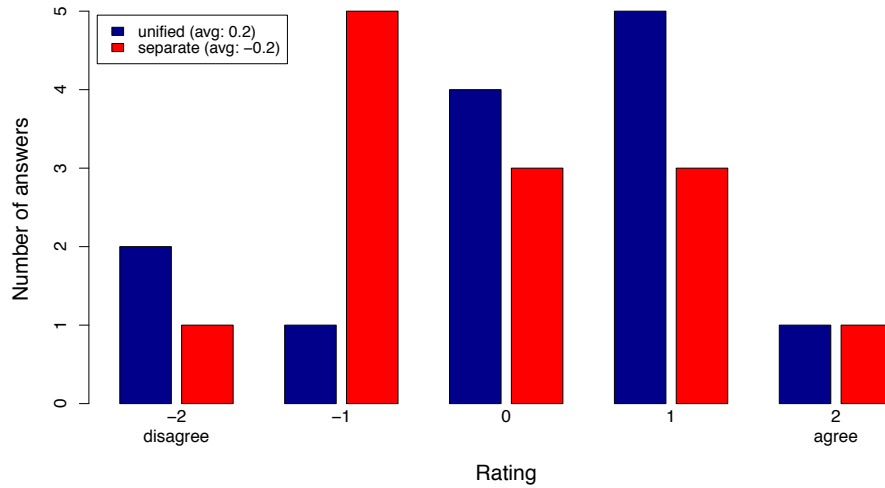


Figure 8.4: Responses to S4: “It was easy to keep scenarios and scenario prototypes consistent.”

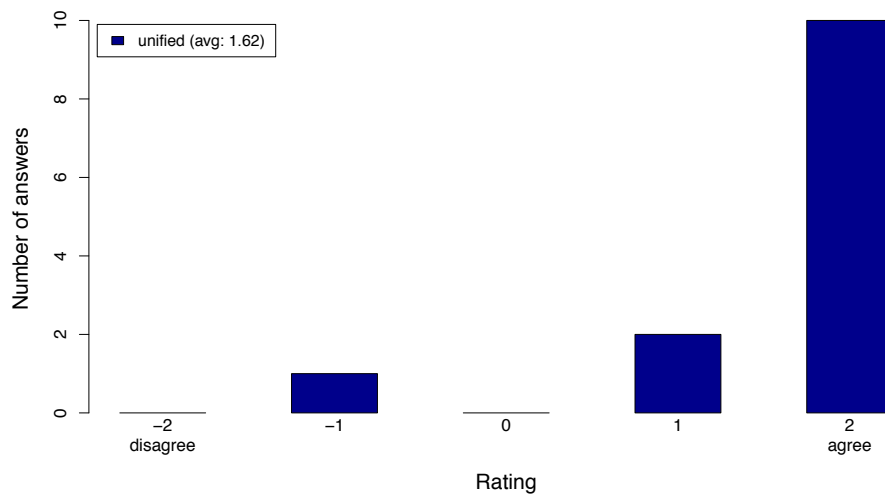


Figure 8.5: Responses to S5: “The error and warning icons motivated me to keep scenarios and scenario prototypes consistent.”

Chapter 9

Conclusion and Future Work

This dissertation explained why a closer collaboration between the fields of human-computer interaction (HCI) and software engineering (SE) is necessary, especially regarding the challenges both fields are confronted with due to the growing importance of mobile devices. The use of scenarios and prototypes, which are techniques that found acceptance in both fields, has been identified as a promising approach. The need for an underlying unified model has been pointed out, as the uncoupled use of scenarios and prototypes bears the risk of diverging artifacts, resulting in inconsistent requirements specifications.

As a solution to this problem, the SCRIPT framework for unified treatment of scenarios and scenario prototypes has been presented, which provides interactivity as well as traceability of requirements. The underlying model has been explained in detail and outlooks on possible extensions of the model have been given. It has been shown how the model can be used in the context of a development project, and how artifacts based on the model can be incorporated into subsequent development steps such as use case specification and identification of analysis model elements. Additionally, it has been shown how the model allows to generate various artifacts such as narrative scenario descriptions, storyboards with textual explanations, and videos depicting the flow of interactions, which can be used for demonstration purposes.

The results of a controlled experiment were presented, which showed that the use of the SCRIPT framework for parallel development of scenarios and scenario prototypes resulted in significantly less errors than compared to uncoupled development.

9.1 Contributions

In the following, the main contributions that have been presented in this dissertation are summarized.

The core contribution of this dissertation is the SCRIPT framework and its underlying model. The SCRIPT model consists of the scenario meta model and interaction meta model, which allow to define scenarios in a semi-structured, technology independent form, and the scenario prototype meta models, which allow to define scenario prototypes for 2D GUIs. As the SCRIPT model unifies the treatment of scenarios and

scenario prototypes, it enables tools to perform automatic checking of consistency between scenarios and scenario prototypes to ensure that they do not become inconsistent during development.

The SCRIPT framework provides traceability from scenarios to scenario prototypes and vice versa via the interaction meta model, which relates interactions described in a scenario to interactions that have been realized in a scenario prototype. The SCRIPT framework makes scenarios and prototypes accessible on a fine-grained level, such as single interactions of a scenario or single user interface elements of a scenario prototype, and thus allows to establish traceability links to other artifacts such as use cases or analysis model elements.

It has been shown how models that are based on the SCRIPT model can be used as the basis for other modeling activities during software development, including the extraction of use cases from the scenario meta model and interaction meta model, the identification of GUI elements from the scenario prototype structural meta model, and the identification of analysis model elements from the scenario prototype structural meta model.

Another contribution is the possibility to generate a variety of artifacts from models that have been created with the SCRIPT framework, and that can be used for information exchange or for discussion with stakeholders to elicit requirements. First, it is possible to generate executable scenario prototypes that convey a feeling about the interaction with the system to stakeholders, which is one of the major goals of the SCRIPT framework. Second, the SCRIPT framework also allows to generate textual documents, which can contain either narrative scenario descriptions, a combination of both narrative scenario descriptions and still images depicting the user interface, or storyboards that visualize the flow of interactions of a scenario with images only. And third, based on the information that is captured with the SCRIPT model, it is possible to automatically generate videos of the interactions between user and system. These may prove especially useful in situations where face-to-face meetings are not possible due to differences in location and/or timezone.

The SCRIPT framework has been evaluated with a controlled experiment, which showed that its use significantly reduces the number of errors that are made when trying to develop scenarios and scenario prototypes in parallel.

9.2 Future Work

In the course of research for this dissertation, some topics have been identified that might benefit from further investigation.

While the scenario meta model and interaction meta model provide a technology independent way of representing the interactions of a scenario, the scenario prototype models focus on the development of 2D GUIs. Both the scenario prototype structural meta model and interaction meta model could be extended to allow for the definition

of additional input and output modalities, like gestures or voice input for user actions, and sound playback or text-to-speech for system actions.

Additionally, in its current state, the SCRIPT framework only supports the creation of scenario prototypes that run on a single device, no matter if it is a desktop computer or a mobile device. A useful extension of the SCRIPT framework would be to allow to define different devices that take part in the execution of a scenario prototype. This would allow to better evaluate multi-user systems as each participant could take part in the evaluation with an individual device. However, this also requires an underlying infrastructure for synchronizing all involved devices during playback.

In this dissertation, the focus was on the creation and documentation of scenarios and scenario prototypes. While the evaluation of those artifacts with stakeholders is of great importance, the details about how feedback from stakeholders can be collected and stored was out of scope. An extension of the SCRIPT framework could provide facilities to gather feedback from stakeholders on scenarios and scenario prototypes that arises during reviews and evaluations, and directly connect it to the underlying model elements that are affected. The dialectic model of questions, options and criteria (QOC) [MYBM91] might prove beneficial for storing the rationale that emerges during reviews and evaluations.

After its deployment, a system only seldom gets used for a longer period of time without any changes necessary. More often than not, the context in which a system is used changes and thus a modification of the system is needed in order to reflect the new situation. If such a modification does not happen, users try to use the system in such a way that they can still solve their tasks at hand, even if the system was not intended to be used that way. As a result, users may be less efficient than it would be possible if the system was adjusted to the changed context. To find out if users actually use the system in ways different than intended from the original specification, the actual system usage could be monitored and compared to the intended, specified usage described with models of the SCRIPT framework. The resulting discrepancy could then be used as an input for further development activities.

Appendix A

Experiment Prototypes Description

This appendix presents a storyboard version of the prototypes that were handed to participants of the experiment described in Chapter 8. Note that participants of the experiments were not handed this storyboard, but they could experience the interactions themselves in the iOS Simulator.

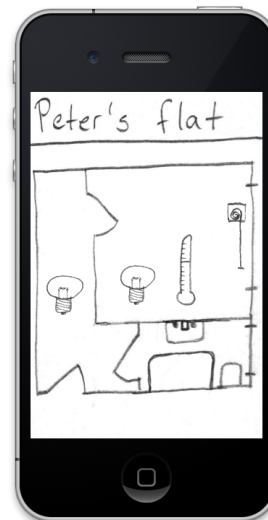
Each image shows the state of the user interface before an interaction takes place. The text underneath it describes the context, if relevant, and which interaction is performed, based on the underlying scenario. The flow of interactions is supposed to be read from left to right and top to bottom.

A.1 Storyboard for Scenario 1: “Peter comes home”

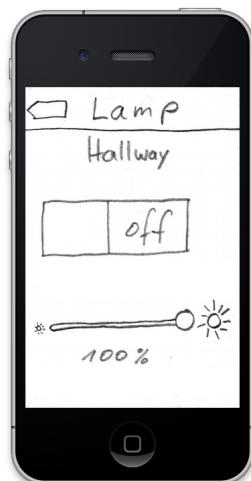
Peter is sitting in the train on his way home from work. Shortly before the train reaches his stop, Peter decides to configure his home devices so that he has a nice welcome. Therefore he takes out his iPhone and starts the Home Automation Controller app.



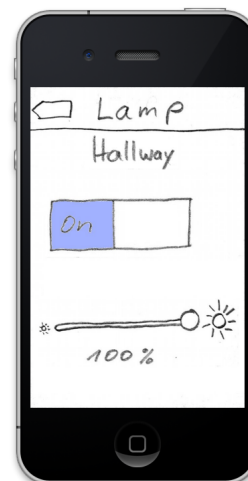
1. On the start screen, he selects the floor plan.



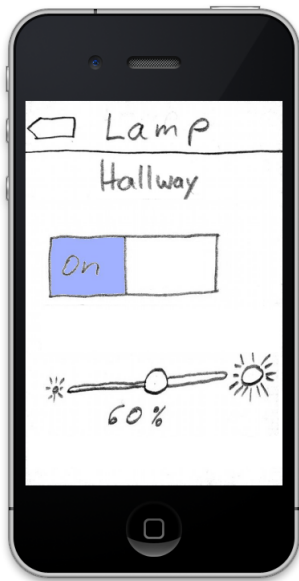
2. The system shows a plan of his flat. Peter selects the hallway light, which is currently off.



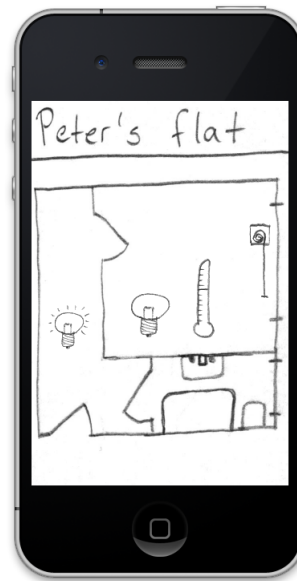
3. Peter switches on the light.



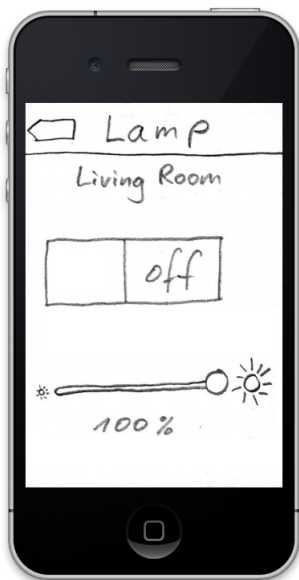
4. To have a more pleasant welcome, he sets the light intensity to 60%.



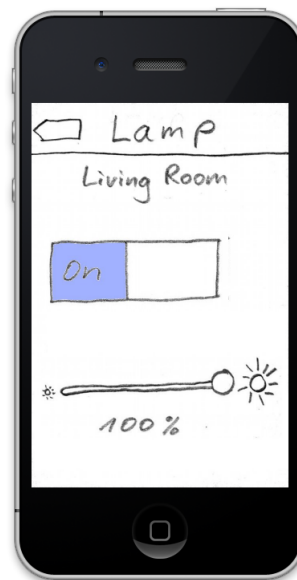
5. Then he taps the back button.



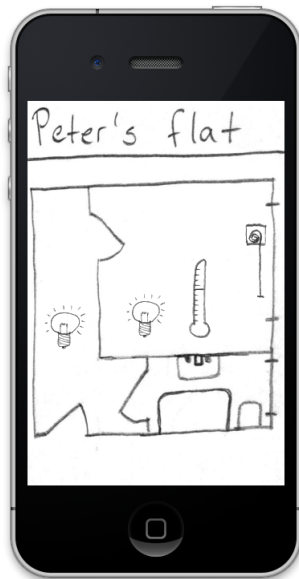
6. The system returns to the floor plan. Peter now selects the light in the living room.



7. The system shows the details of the living room light. Peter switches the light on.



8. Then he taps the back button. The system returns to the floor plan.



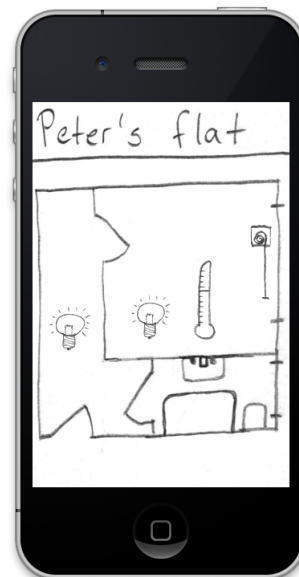
9. Peter now selects the temperature setting.



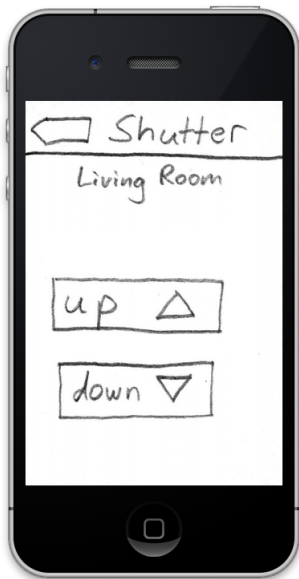
10. The system shows the temperature details, with the current temperature being 18 degrees. Peter sets the temperature to 22 degrees.



11. Then he taps the back button.



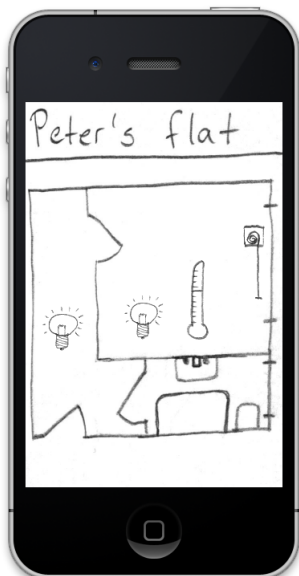
12. The system returns to the floor plan. Last, he selects the shutter.



13. The system shows the details for the shutter. Peter commands the shutter to close.



14. The system displays that the shutter is now closing. Peter taps on the back button.



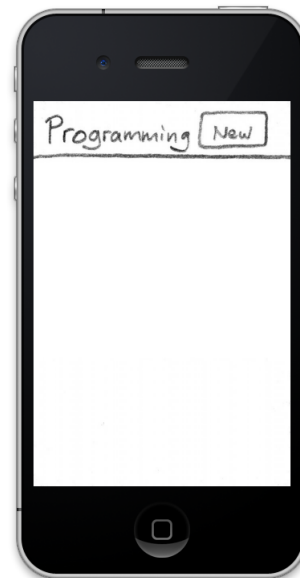
15. The system returns to the floor plan. Peter is satisfied with his settings. He closes the app and puts the iPhone back into his pocket.

A.2 Storyboard for Scenario 2: “Peter programs”

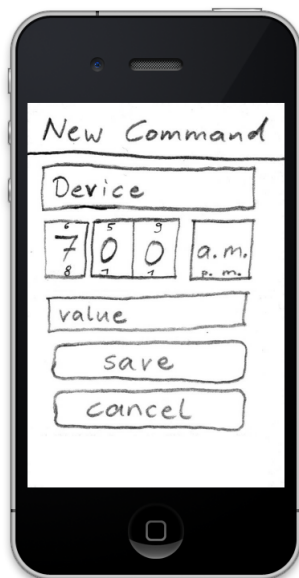
Peter is sitting on the couch and decides to program the shutter. He wants it to open in the morning and close in the evening. Therefore he takes out his iPhone and starts the Home Automation Controller app.



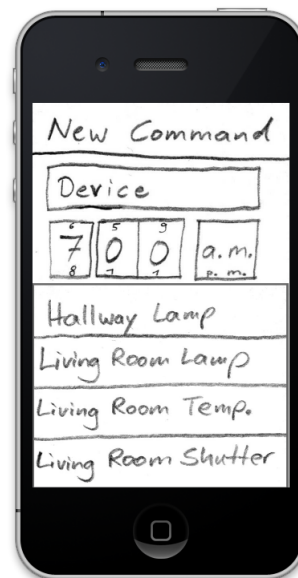
1. On the start screen, he selects “Programming”.



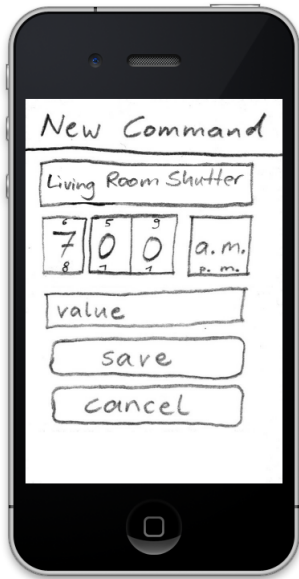
2. The system shows an empty list. Peter selects to add a new command.



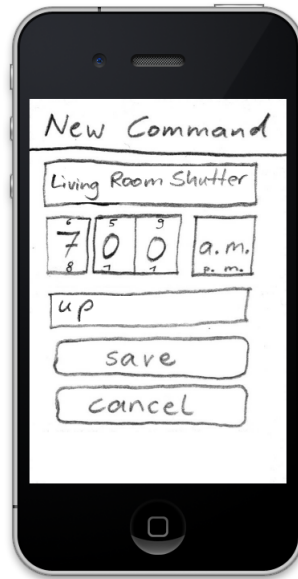
3. The system shows the screen for defining automatic commands. Peter selects the “device” field.



4. The system shows a dialog with all available devices by name. Peter selects “Living Room Shutter”.



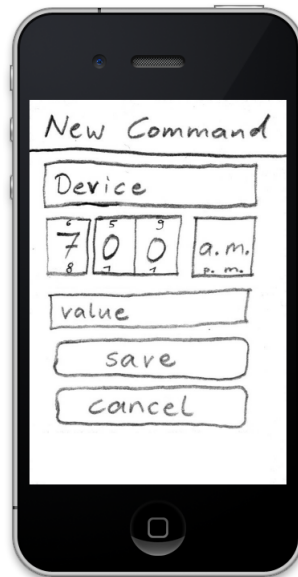
5. The system hides the dialog and sets the device to "Living Room Shutter". In the field for the value of the command, Peter enters "up".



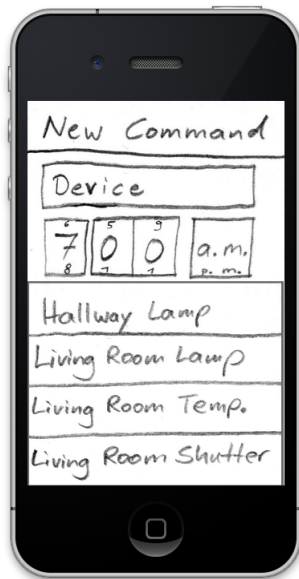
6. Then, he saves the command. The system returns to the list of commands, where the new command now appears.



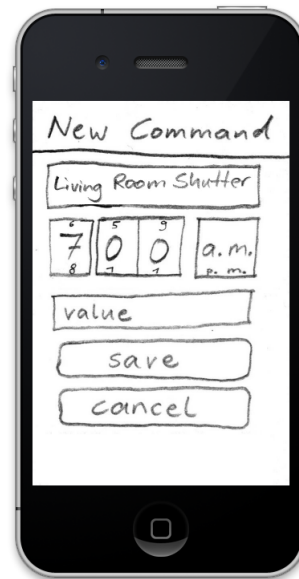
7. Peter selects to add another command.



8. The system again shows the screen for defining new commands. Peter selects the "device" field.



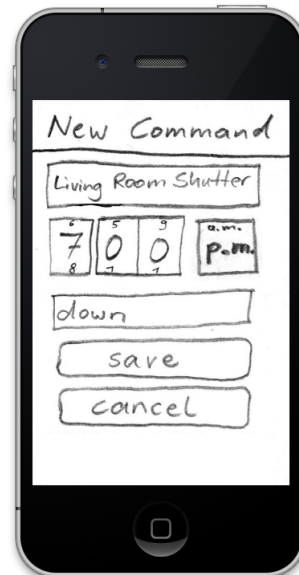
9. The system shows a dialog with all available devices by name. Peter selects “Living Room Shutter”.



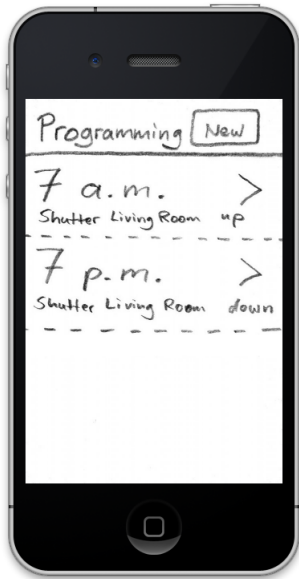
10. The system hides the dialog and sets the device to “Living Room Shutter”. Peter changes the time of day to 7 p.m.



11. In the field for the value of the command, Peter enters “down”.



12. Then, he saves the second command.



13. The system returns to the list of commands, where now both commands appear. Peter is satisfied, closes the app and puts away his iPhone.

Appendix B

Experiment Work Package Descriptions

In this appendix, the work package descriptions that were given to the participants of the experiment are reprinted.

Each work package description starts with a short motivational text that explains why the change is requested and gives a textual summary of the changes. The largest part of the work package description consists of scene images on the left hand side and textual scenario descriptions on the right side. Both images and scenario text describe how the new flow of interactions should be after the participants applied the necessary changes. In order to make it easier for the participants to find the position in the prototype where the changes start, the first image of each work package depicts an already existing scene from which on the changes should be applied.

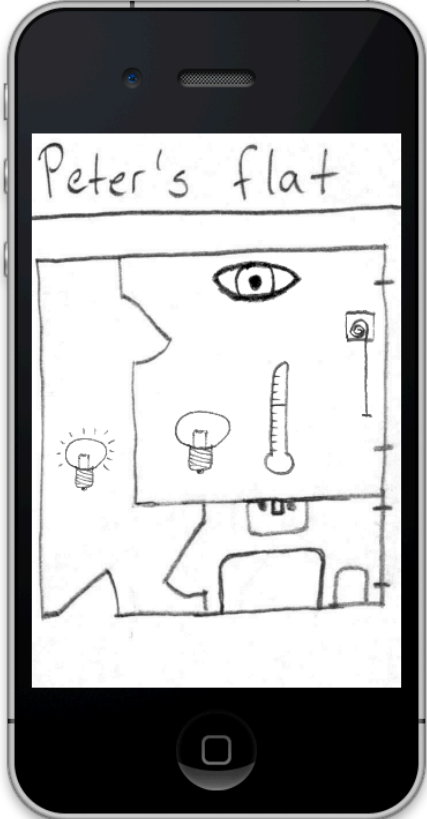
Each work package explicitly requested to modify both prototypes and scenarios so that they stay consistent. Only participants in the unified group, however, used the SCRIPT Editor where both the scenario and prototype editing component were enabled, and which automatically warned them whenever scenarios and prototypes were no longer consistent.

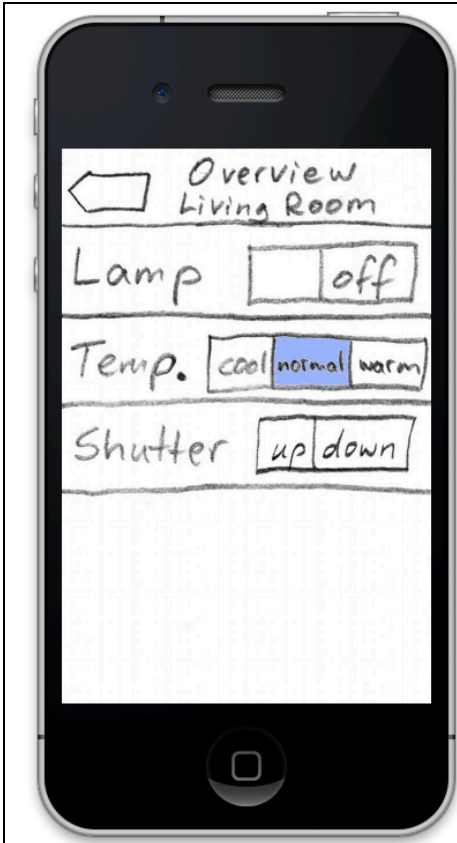
Dear developers,

Upon inspection of the prototype for the first scenario, we realized that the sequential operation of each single device is a bit tedious. Therefore we would like to test an alternative where all devices in the living room are accessible at once via a list view.

We already setup scenario 3 and prototype 3 as an alternative flow of events. Please extend them with the interactions described below. The initial steps, where Peter switches on the hallway light, should stay the same.

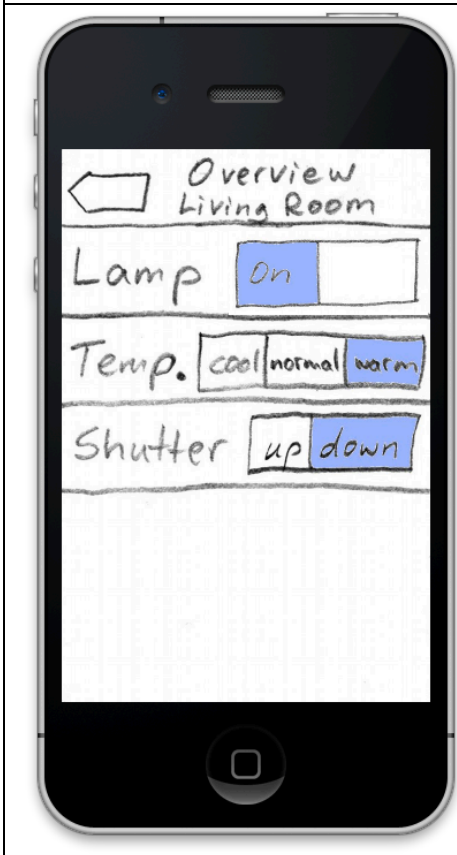
As you can see, the living room devices are no longer operated one after another, but rather all at once. All images that you might need are available in the screens and snippets folders.

[...]	All previous steps stay the same.
	<p>This is the floor plan after Peter turned on the hallway light. Note that the icon for the hallway light is lit.</p> <p>Peter taps on the overview icon for the living room.</p>

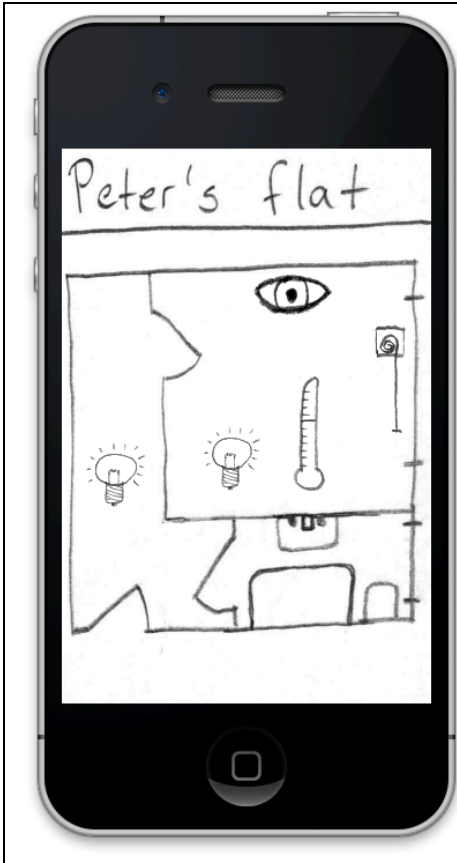


The system shows the overview screen.

On the overview screen, Peter taps on the lamp slider to turn on the living room lamp. Then he commands the shutter to close.



After he made all settings, Peter taps on the back button.



The system returns to the floor plan. The floor plan now also shows that the living room light has been lit.

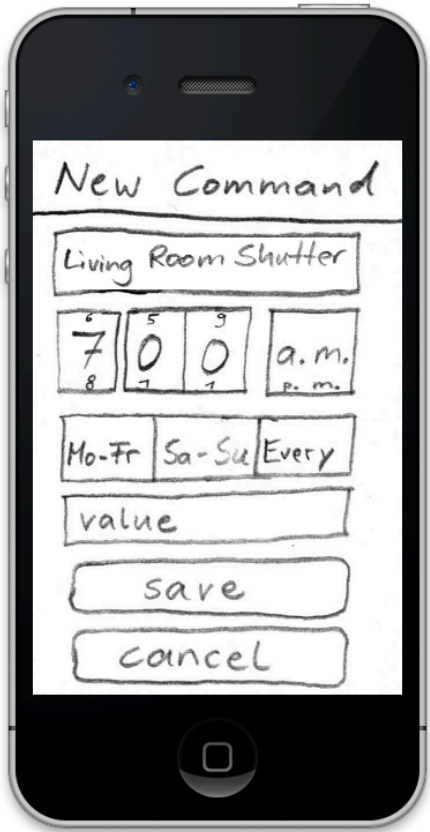
Note that the icons for temperature and shutter **do not** change their appearance depending on their state.

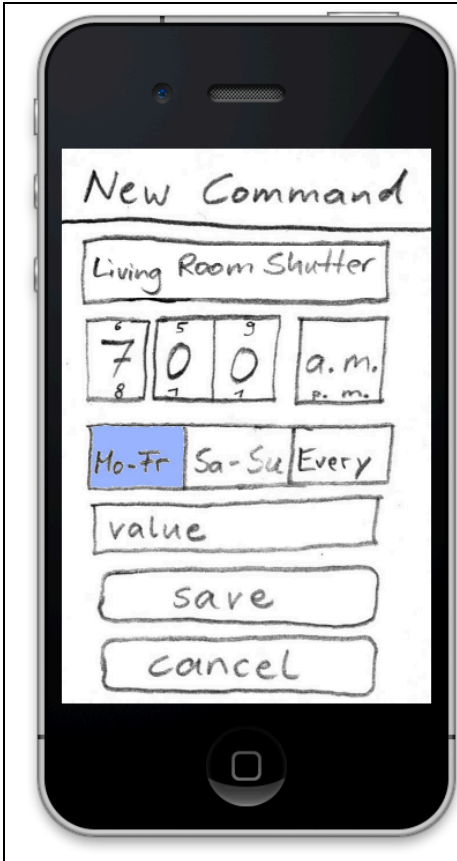
Dear developers,

For the device programming part, we need to do a modification to the command details screen. We showed the prototype to customers and many of them mentioned the wish for being able to select whether the command should be executed every Monday to Friday, on Saturday and Sunday, or everyday.

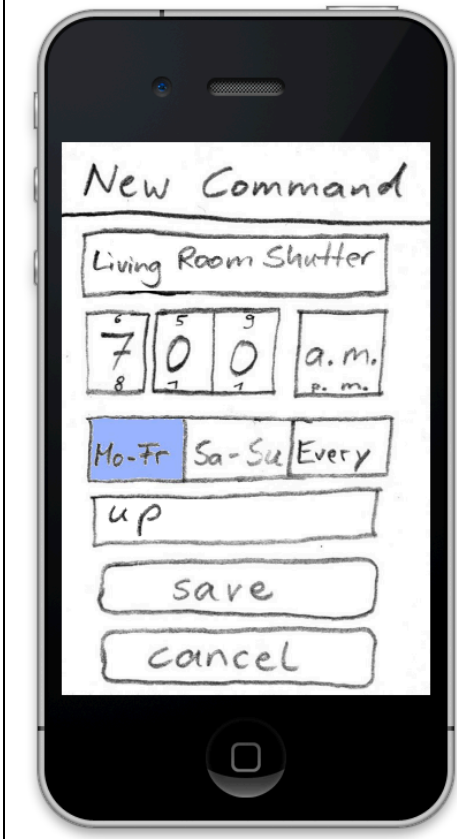
Attached you can find an excerpt from the changed sequence of scenes for the prototype. The remaining steps should stay the same.

Please update scenario 2 and prototype 2 accordingly and let Peter select the "Mo - Fr" option for both the "up" and "down" commands.

[...]	All previous steps stay the same.
	Upon programming the first command, Peter selected "Living Room Shutter" as device. This results in the scene to the left. Peter selects "Mo-Fr" as frequency for the new command.



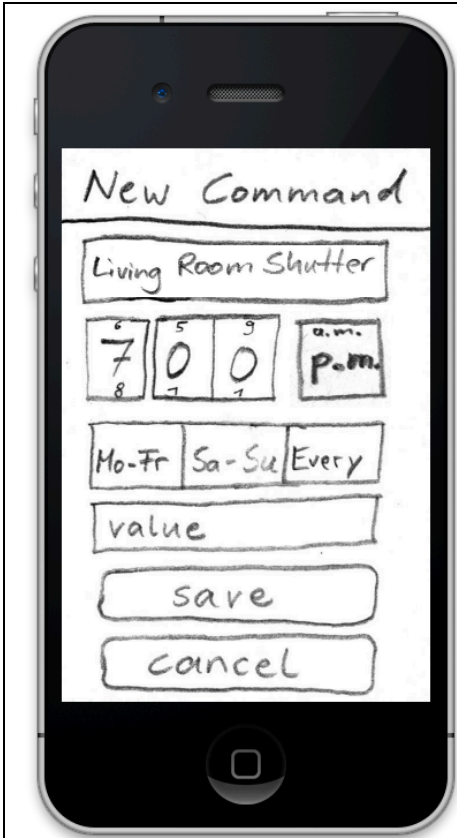
In the field for the value of the command, Peter enters "up".



Then, he saves the command.

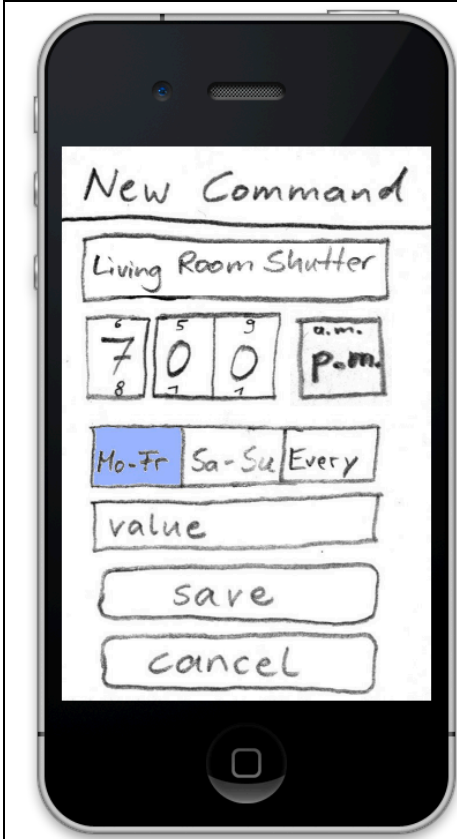
[...]

The intermediate steps stay the same.

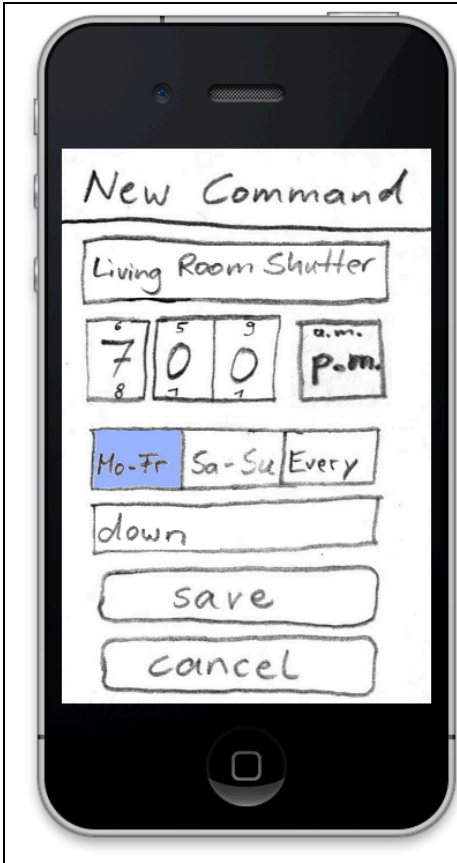


Peter selected "Living Room Shutter" as device and "p.m." as day of time for the second command.

Peter selects "Mo-Fr" as frequency for the new command.



In the field for the value of the command, Peter enters "down".



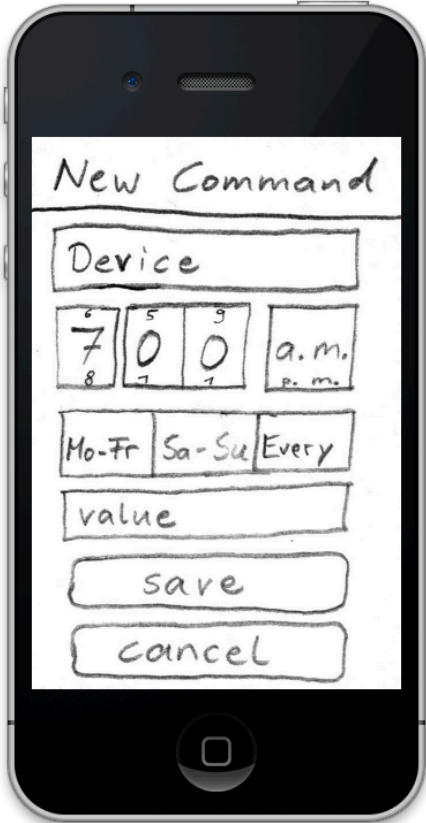
Then, he saves the second command.

The system returns to the list of commands, where now both commands appear.

Dear developers,

We have a need for change in the programming scenario 2: Our usability experts determined that the selection of the desired device from a list by name is not very user friendly. Instead, the system should navigate to the floor plan and let the user select the desired device from there.

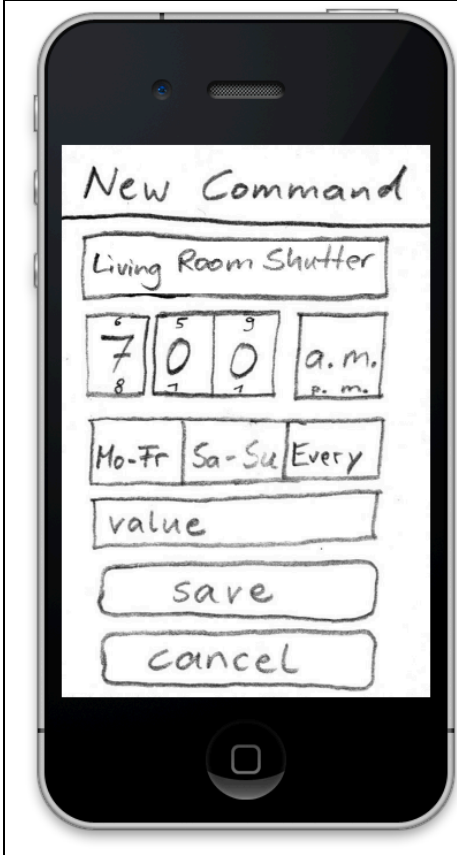
Please adjust scenario 2 and prototype 2 according to the example sequence of scenes below.

[...]	All previous steps stay the same.
	<p>Peter taped on "New" in order to create a new command. The system shows the screen for defining automatic commands.</p> <p>Peter selects the "device" field.</p>



The system shows the floorplan with all available devices.

Peter selects the living room shutter.



The system returns to the new command screen, where "Living Room Shutter" has been set as device.

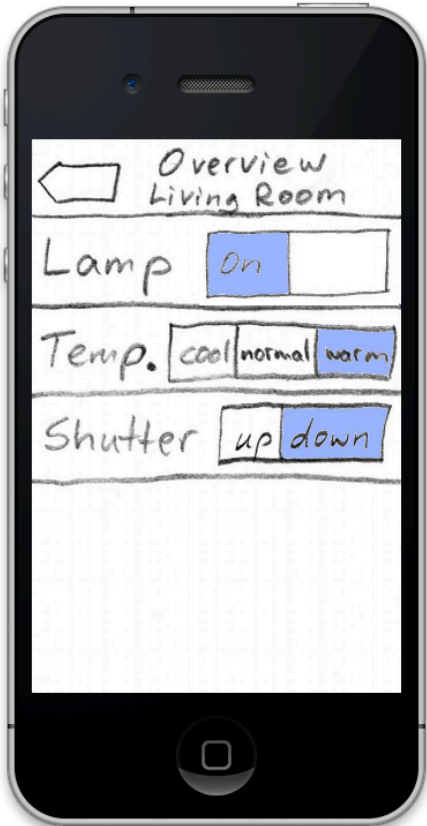
The remaining flow of events stays the same. The programming of the second command should also use the floorplan for device selection.

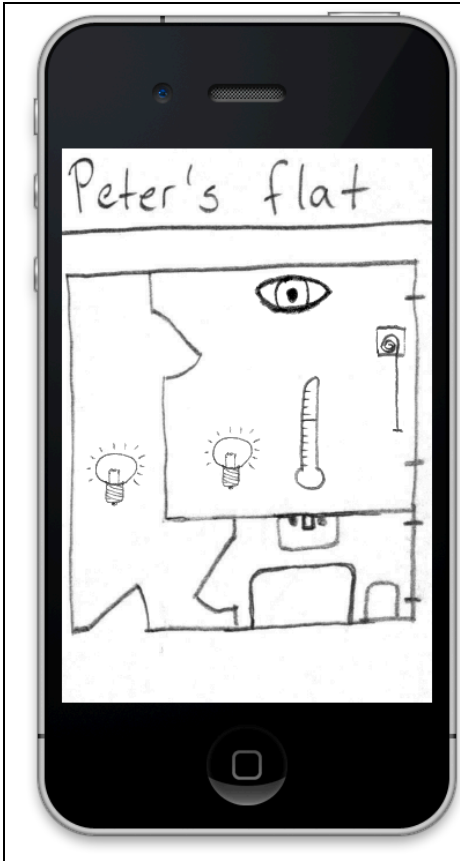
Dear developers,

Many customers felt uneasy about the part in the scenario where Peter closes the shutter from the overview screen while he is not at home. Some of them argued that they have flower boxes on their window sills and that they are afraid that their flowers might get hurt when the shutter closes while they are not at home.

Additionally, our marketing department noted that it seems a bit strange that Peter dims the hallway light. They would prefer our prototype to allow dimming the living room light instead of the hallway light, as this makes up a much better story.

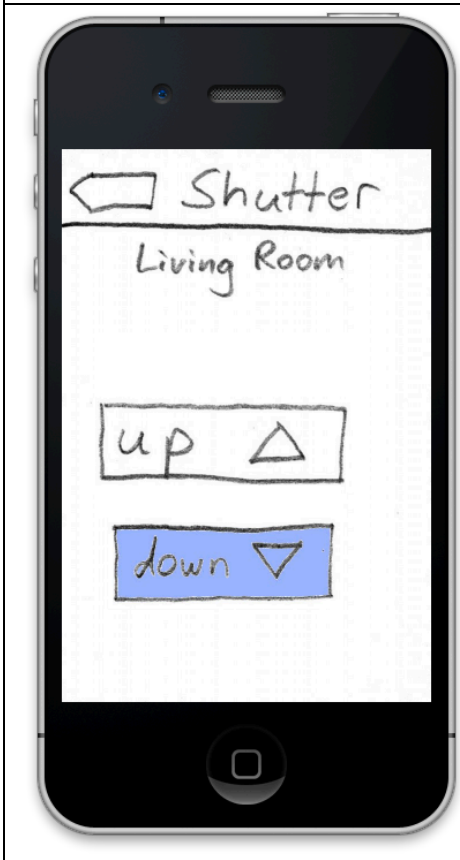
Therefore we request you to alter scenario 3 and prototype 3. Please find below the sequence of interactions for the added interactions.

[...]	
	<p>Peter selected the overview from the floorplan. He switched the lamp on, set the temp to warm and commanded the shutter to close.</p> <p>After he made all settings, Peter taps on the back button.</p>



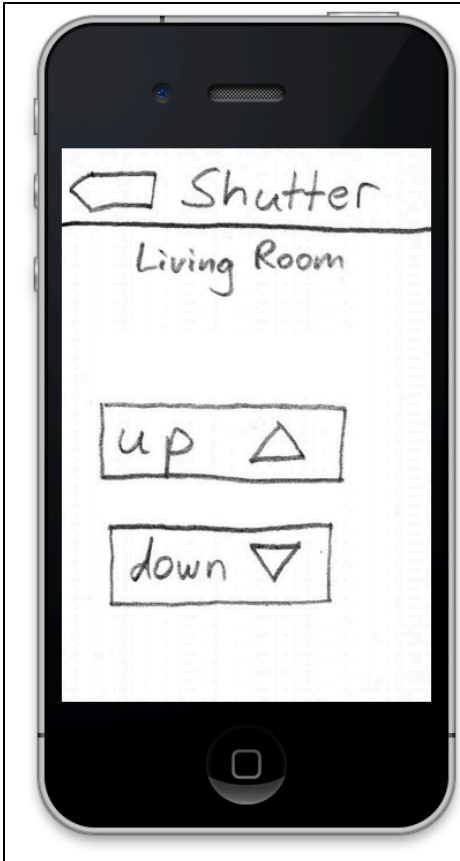
The system returns to the floorplan.

Back on the floor plan, he realizes that he has a flower box on the window sill. In order to prevent it from getting hurt, he decides to stop the shutter again. Therefore he selects the shutter.



The system shows the shutter screen, where the shutter is marked as currently going down.

Peter taps on "down" in order to stop the shutter.



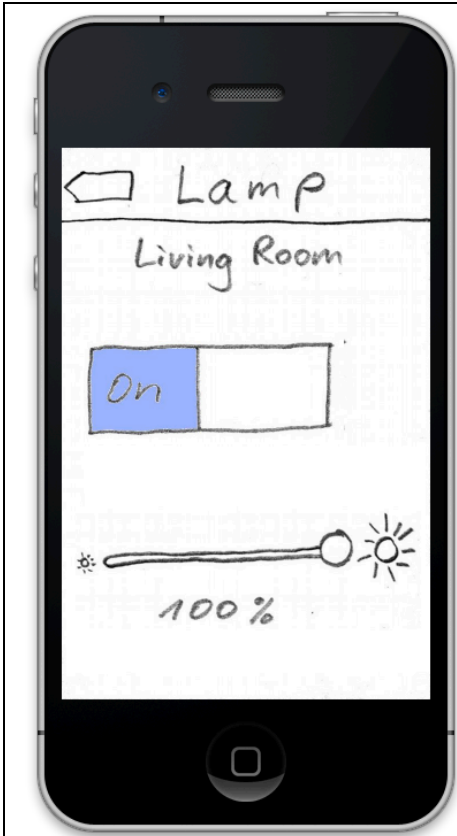
The system shows that the shutter has stopped.

Peter taps the back button.



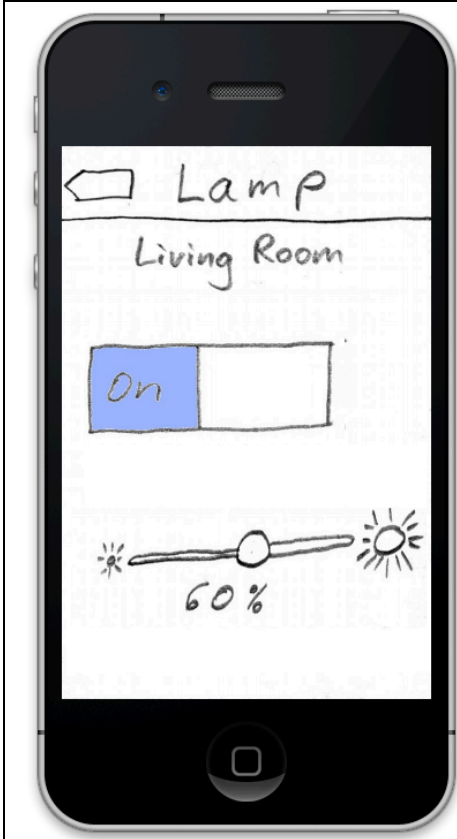
The system shows the floor plan again.

Peter wants to lower the intensity of the living room light in order to have a more pleasant welcome. Therefore he selects the living room light.



The system shows the details of the living room light.

Peter sets the intensity to 60%.



The system shows that the intensity has been lowered.

Peter taps on the back button. The system shows the floor plan again.

List of Figures

2.1	Mental models of designer and user. Adapted from [Nor02].	11
2.2	Mental models and system models	13
4.1	Relation between <i>number of design alternatives</i> and <i>creation effort per alternative</i>	29
4.2	Interplay between <i>mind</i> and <i>sketch</i> . Adapted from [Bux07]	30
4.3	Continuum between Sketch and Prototype. Adapted from [Bux07]	31
4.4	Aspects of prototyping by Houde and Hill. From [HH97]	34
4.5	Wireframe of the web page of the Chair for Applied Software Engineering	37
5.1	Example of a screen generated with the <i>Screen-Based Scenario Generator</i> . From [HY88]	46
5.2	Overview of the SCRIPT model, which consists of the scenario meta model, the interaction meta model, the scenario prototype structural meta model and the scenario prototype interaction meta model	47
5.3	Scenario meta model and interaction meta model	48
5.4	Possible instantiations of the scenario meta model and interaction meta model	50
5.5	Scenario prototype structural meta model	52
5.6	Example of GUIElementImage without (a) and with (b) scrolling enabled	54
5.7	Example of layering in a GUIScene. Part (a) shows the resulting scene, part (b) shows the division into single GUIElementAppearances	55
5.8	Complete model of SCRIPT	56
6.1	Overview of the activities involved for applying the SCRIPT framework	64
6.2	Orders of SCRIPT model traversal	66
6.3	Relation between scenario meta mode, interaction meta model, and use case meta model	70
6.4	Relation between scenario prototype structural meta model and user interface meta model	71
6.5	Roles of user interface elements	72
6.6	Relation between role of an user interface element and analysis meta model	73
6.7	Example for identification of attributes, classes and associations	74
7.1	The SCRIPT Editor	80

7.2	The Scenario Prototype Editor	81
7.3	The Scenario Prototype Editor, showing an inconsistency between scenario prototype and scenario, which is displayed with an error icon in the Scenario Prototype Editor and a warning icon in the Navigator	82
7.4	The Validation view, listing an inconsistency between scenario prototype and scenario, which is shown as a validation error	83
7.5	The Scenario Editor	84
7.6	Layers of the SCRIPT Editor	85
7.7	Components of the SCRIPT evaluation setup	86
8.1	Responses to S1: “The scenarios help me in understanding the scenario prototypes in the preparation phase.”	96
8.2	Responses to S2: “I like the idea of using both scenarios and scenario prototypes.”	97
8.3	Responses to S3: “The SCRIPT Editor is usable.”	97
8.4	Responses to S4: “It was easy to keep scenarios and scenario prototypes consistent.”	98
8.5	Responses to S5: “The error and warning icons motivated me to keep scenarios and scenario prototypes consistent.”	98

Bibliography

- [AAB07] Jonathan Arnowitz, Michael Arent, and Nevin Berger. *Effective prototyping for software makers*. Elsevier, 2007.
- [Abb83] Russell J Abbott. Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894, November 1983. ACM ID: 358441.
- [Abb87] Russell J Abbott. Knowledge abstraction. *Commun. ACM*, 30(8):664–671, August 1987. ACM ID: 27652.
- [ABJ05] Rob J. Adams, Len Bass, and Bonnie E. John. Experience with using general usability scenarios on the software architecture of a collaborative system. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 87–112. Springer-Verlag, Berlin/Heidelberg, 2005.
- [AH93] S. Asur and S. Hufnagel. Taxonomy of rapid-prototyping methods and tools. In , *Fourth International Workshop on Rapid System Prototyping, 1993. Shortening the Path from Specification to Prototype. Proceedings*, pages 42–56. IEEE, June 1993.
- [Ala84] Maryam Alavi. An assessment of the prototyping approach to information systems development. *Commun. ACM*, 27(6):556–563, June 1984.
- [All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.
- [Als05] T. A Alspaugh. Temporally expressive scenarios in ScenarioML. *Institute for Software Research Technical Report UCI-ISR-05*, 6, 2005.
- [Ana12] Strategy Analytics. Android captures record 39 percent share of global tablet shipments in q4 2011. Technical report, Bosten, MA, USA, January 2012.
- [And89] S. J Andriole. *Storyboard Prototyping a New Approach to User Requirements Analysis*. QED Information Sciences Inc., Wellesley, Mass, 1989.

- [And94] S.J. Andriole. Fast, cheap requirements: prototype, or else! *Software, IEEE*, 11(2):85–87, 1994.
- [App11] Apple. Xcode 4 user guide: Designing user interfaces in xcode. <https://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/InterfaceBuilder/InterfaceBuilder.html>, October 2011.
- [App12a] Apple. Apple reports first quarter results. Technical report, Cupertino, January 2012.
- [App12b] Apple. Apple’s app store downloads top 25 billion. Technical report, Cupertino, March 2012.
- [ATB06] Thomas A. Alspaugh, Bill Tomlinson, and Eric Baumer. Using social agents to visualize software scenarios. In *Proceedings of the 2006 ACM symposium on Software visualization - SoftVis ’06*, page 87, Brighton, United Kingdom, 2006.
- [BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
- [BBL⁺02] Jakob Bardram, Claus Bossen, Andreas Lykke-Olesen, Rune Nielsen, and Kim Halskov Madsen. Virtual video prototyping of pervasive healthcare systems. In *Proceedings of the 4th conference on Designing interactive systems: processes, practices, methods, and techniques, DIS ’02*, page 167–177, London, England, 2002. ACM. ACM ID: 778738.
- [BBLZ96] D. Bäumer, W. Bischofberger, H. Lichter, and H. Züllighoven. User interface prototyping-concepts, tools, and experience. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 532–541, 1996.
- [BD09] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 3 edition, August 2009.
- [Bin99] Thomas Binder. Setting the stage for improvised video scenarios. In *CHI ’99 extended abstracts on Human factors in computing systems - CHI ’99*, page 230, Pittsburgh, Pennsylvania, 1999.
- [BJ04] Kai Blankenhorn and Mario Jeckle. A UML profile for GUI layout. In Mathias Weske and Peter Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies*, volume 3263, pages 110–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

- [BJHW00] M. Brockmeyer, F. Jahanlan, C. Heitmeyer, and E. Winner. A flexible, extensible simulation environment for testing real-time specifications. *Computers, IEEE Transactions on*, 49(11):1184–1201, 2000.
- [BKC01] Brian P. Bailey, Joseph A. Konstan, and John V. Carlis. DEMAIS: designing multimedia applications with interactive storyboards. In *Proceedings of the ninth ACM international conference on Multimedia, MULTIMEDIA '01*, page 241–250, New York, NY, USA, 2001. ACM.
- [Blo05] Stefan Blomkvist. Towards a model for bridging agile development and User-Centered design. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 219–244. Springer-Verlag, Berlin/Heidelberg, 2005.
- [Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, New Jersey, November 1981.
- [Boe88] B. W Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.
- [Boe00] B. Boehm. Requirements that handle IKIWISI, COTS, and rapid change. *Computer*, 33(7):99–102, 2000.
- [BPTM03] J. Belenguer, J. Parra, I. Torres, and P. J Molina. HCI designers and engineers: It is possible to work together? *CLOSING THE GAPS: Software Engineering and Human-Computer Interaction*, 2003.
- [Bux07] Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
- [Car95] J. M Carroll. *Scenario-based design: envisioning work and technology in system development*. John Wiley and sons, New York, NY, USA, 1995.
- [Car97] Jim A. Carter. Putting usability first in the design of web sites. Toronto, Ontario, Canada, November 1997.
- [Chi90] Mark H. Chignell. A taxonomy of user interface terminology. *SIGCHI Bull.*, 21(4):27, 1990.
- [CLSF05] Jim A. Carter, Jun Liu, Kevin Schneider, and David Fourney. Transforming usability engineering requirements into software engineering specifications: From PUF to UML. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating*

- Usability in the Software Development Lifecycle*, volume 8, pages 147–169. Springer-Verlag, Berlin/Heidelberg, 2005.
- [COB06] Oliver Creighton, Martin Ott, and Bernd Bruegge. Software Cinema-Video-based requirements engineering. *Requirements Engineering, IEEE International Conference on*, 0:109–118, 2006.
- [Coc01] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Boston, MA, USA, 2001.
- [Con95] Larry L. Constantine. Essential modeling: use cases for user interfaces. *interactions*, 2(2):34–46, April 1995.
- [CR92] John M. Carroll and Mary Beth Rosson. Getting around the task-artifact cycle: how to make claims and design by scenario. *ACM Transactions on Information Systems*, 10:181–212, April 1992.
- [CRCK98] J.M. Carroll, M.B. Rosson, G. Chin, and J. Koenemann. Requirements development in scenario-based design. *IEEE Transactions on Software Engineering*, 24:1156–1170, December 1998.
- [Cre05] Oliver Creighton. *Software Cinema: Employing Digital Video in Requirements Engineering*. PhD thesis, Technische Universität München, München, June 2005.
- [CU93] Bay-Wei Chang and David Ungar. Animation: from cartoons to the user interface. In *Proceedings of the 6th annual ACM symposium on User interface software and technology, UIST '93*, page 45–55, New York, NY, USA, 1993. ACM.
- [DDN92] Sarah Douglas, Eckehard Doerry, and David Novick. QUICK: a tool for graphical user-interface construction by non-programmers. *The Visual Computer*, 8(2):117–133, March 1992.
- [DH01] Werner Damm and David Harel. LSCs: breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.
- [DJA93] N. Dahlbäck, A. Jönsson, and L. Ahrenberg. Wizard of oz studies — why and how. *Knowledge-Based Systems*, 6(4):258–266, December 1993.
- [dSP03] P. P da Silva and N. W Paton. User interface modeling in UMLi. *IEEE Software*, 20(4):62–69, August 2003.
- [Ecl12a] Eclipse. Eclipse modeling framework project. <http://www.eclipse.org/modeling/emf/>, April 2012.

- [Ecl12b] Eclipse. EMF client platform. <http://eclipse.org/emfclient/>, May 2012.
- [Ecl12c] Eclipse. EMFStore project home. <http://eclipse.org/emfstore/>, April 2012.
- [Ecl12d] Eclipse. Graphiti. <http://eclipse.org/graphiti/>, May 2012.
- [Ecl12e] Eclipse. Rich client platform. <http://www.eclipse.org/rcp/>, April 2012.
- [EK00] Mohammed Elkoutbi and Rudolf K. Keller. User interface prototyping based on UML scenarios and High-Level petri nets. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825, pages 166–186. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
- [EKK99] M. Elkoutbi, I. Khriiss, and R. K Keller. Generating user interface prototypes from scenarios. pages 150–158, 1999.
- [EKK06] Mohammed Elkoutbi, Ismaïl Khriiss, and Rudolf K. Keller. Automated prototyping of user interfaces based on UML scenarios. *Automated Software Engineering*, 13(1):5–40, January 2006.
- [Fel12] Albert Feller. Evaluation and development of a prototyping tool for mobile graphical user interfaces: a practical view. Master’s thesis, Technische Universität München, Munich, Germany, March 2012.
- [FJM05] Xavier Ferre, Natalia Juristo, and Ana M. Moreno. Which, when and how usability techniques and activities should be integrated. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 173–200. Springer-Verlag, Berlin/Heidelberg, 2005.
- [Flo86] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*. Springer-Verlag GmbH, July 1986.
- [Gar12] Gartner. Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth. <http://www.gartner.com/it/page.jsp?id=1924314>, February 2012.
- [GC04] Kentaro Go and John M. Carroll. The blind men and the elephant. *interactions*, 11(6):44–53, November 2004.
- [GFHR95] P. A Gough, F. T Fodemski, S. A Higgins, and S. J Ray. Scenarios-an industrial case study and hypermedia enhancements. In , *Proceedings of the Second IEEE International Symposium on Requirements Engineering, 1995*, pages 10– 17. IEEE, March 1995.

- [GGB⁺05] Jan Gulliksen, Bengt Göransson, Inger Boivie, Jenny Persson, Stefan Blomkvist, and Åsa Cajander. Key principles for User-Centred systems design. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 17–36. Springer-Verlag, Berlin/Heidelberg, 2005.
- [GGR93] J. Grabowski, P. Graubmann, and E. Rudolph. The standardization of message sequence charts. In *Software Engineering Standards Symposium, 1993. Proceedings., 1993*, pages 48–63, September 1993.
- [GGS09] G. Gabrysiak, H. Giese, and A. Seibel. Interactive visualization for elicitation and validation of requirements with Scenario-Based prototyping. In *Requirements Engineering Visualization (REV), 2009 Fourth International Workshop on*, pages 41–45, 2009.
- [GGS11] Gregor Gabrysiak, Holger Giese, and Andreas Seibel. Towards next generation design thinking: Scenario-Based prototyping for designing complex software systems with multiple users. In Christoph Meinel, Larry Leifer, and Hasso Plattner, editors, *Design Thinking, Understanding Innovation*, pages 219–236. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-13757-0_13.
- [GHO02] Sallie Gregor, Joseph Hutson, and Colleen Oresky. Storyboard process to assist in requirements verification and adaptation to capabilities inherent in COTS. In John Dean and Andrée Gravel, editors, *COTS-Based Software Systems*, volume 2255 of *Lecture Notes in Computer Science*, pages 132–141. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45588-4_13.
- [GM03] L. Gorlenko and R. Merrick. No wires attached: Usability challenges in the connected mobile world. *IBM Systems Journal*, 42(4):639–651, 2003.
- [Goo12] Google. WindowBuilder user guide. <http://code.google.com/intl/de-DE/javadevtools/wbpro/>, March 2012.
- [Gra94] Ian Graham. *Object-oriented methods*. Addison-Wesley, 1994.
- [Gro06] Object Management Group. UML diagram interchange. <http://www.omg.org/spec/UMLDI/1.0/>, April 2006.
- [Gro12a] Object Management Group. Business process model and notation. <http://www.bpmn.org/>, May 2012.
- [Gro12b] Object Management Group. Object management group - UML. <http://www.uml.org/>, February 2012.

- [HB95] Karen Holtzblatt and Hugh R Beyer. Requirements gathering: the human factor. *Commun. ACM*, 38(5):31–32, 1995. ACM ID: 203361.
- [HBC⁺96] Thomas Hewett, Ronald Baecker, Stuart Card, Tom Carey, Jean Gasen, Marylin Mantei, Gary Perlman, Gary Strong, and William Verplank. ACM SIGCHI curricula for Human-Computer interaction. <http://old.sigchi.org/cdg/cdg2.html>, 1996.
- [HCR05] Steven R. Haynes, John M. Carroll, and Mary Beth Rosson. Integrating User-Centered design knowledge with scenarios. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 269–286. Springer-Verlag, Berlin/Heidelberg, 2005.
- [Her03] Morten Hertzum. Making use of scenarios: a field study of conceptual design. *International Journal of Human-Computer Studies*, 58(2):215–239, February 2003.
- [HH97] Stephanie Houde and Charles Hill. What do prototypes prototype? In M Helander, T Landauer, and P Prahbu, editors, *Handbook of Human-Computer Interaction*. Elsevier Science B. V, Amsterdam, 1997.
- [HKLB98] Constance L Heitmeyer, James Kirby, Bruce G Labaw, and Ramesh Bharadwaj. SCR*: a toolset for specifying and analyzing software requirements. In *Proceedings of the 10th International Conference on Computer Aided Verification, CAV '98*, page 526–531, London, UK, 1998. Springer-Verlag. ACM ID: 733627.
- [HM03] David Harel and Rami Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and Systems Modeling*, 2(2):82–107, July 2003.
- [HS06] Andreas Holzinger and Wolfgang Slany. XP + UE -> XU praktische erfahrungen mit eXtreme usability. *Informatik-Spektrum*, 29(2):91–97, February 2006.
- [HSG⁺94] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis. *IEEE Software*, 11(2):33–41, March 1994.
- [Hum89] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley Professional, January 1989.
- [HVF03] Morten Borup Harning, Jean Vanderdonckt, and Murielle Florins. Closing the gaps: Software engineering and Human-Computer interaction. Zurich, Switzerland, September 2003.

- [HY88] P. Hsia and A. T. Yaung. Screen-Based scenario generator: a tool for scenario-based prototyping. In *Software Track, Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, 1988. Vol. II*, volume 2, pages 455–461. IEEE, January 1988.
- [IEE97] IEEE. IEEE standard 1074 for developing software life cycle processes, 1997.
- [ISO10] ISO. ISO 9241-210:2010 human-centred design for interactive systems, 2010.
- [JBKC04] Bonnie E. John, Len Bass, Rick Kazman, and Eugene Chen. Identifying gaps between HCI, software engineering, and design, and boundary objects to bridge them. page 1723. ACM Press, 2004.
- [JC07] Jung-Sing Jwo and Yu Chin Cheng. Pseudo software: a new concept for iterative requirement development and validation. In *Asia-Pacific Software Engineering Conference*, volume 0, pages 105–111, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [JC10] Jung-Sing Jwo and Yu Chin Cheng. Pseudo software: A mediating instrument for modeling software requirements. *Journal of Systems and Software*, 83(4):599–608, April 2010.
- [JK05] Bill Jerome and Rick Kazman. Surveying the solitudes: An investigation into the relationships between human computer interaction and software engineering in practice. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 59–70. Springer-Verlag, Berlin/Heidelberg, 2005.
- [JL08] Kasper Løvborg Jensen and Lars Bo Larsen. The challenge of evaluating the mobile and ubiquitous user experience. Sydney, Australia, 2008.
- [Jok05] Timo Jokela. Guiding designers to the world of usability: Determining usability requirements through teamwork. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 127–145. Springer-Verlag, Berlin/Heidelberg, 2005.
- [Kar97] John Karat. Evolving the scope of user-centered design. *Commun. ACM*, 40(7):33–38, July 1997.
- [KBB03] Rick Kazman, Len Bass, and Jan Bosch. Bridging the gaps between software engineering and Human-Computer interaction. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, page 777–778, Washington, DC, USA, 2003. IEEE Computer Society.

- [KJ02] Hermann Kaindl and Rudolf Jezek. From usage scenarios to user interface elements in a few steps. In *Computer-Aided Design of User Interfaces III*, Computer-Aided Design of User Interfaces. Kluwer Academic Publishers, Dordrecht, Netherlands, 2002.
- [KNS⁺08] Tommi Kärkkäinen, Miika Nurminen, Panu Suominen, Tuomo Pieniluoma, and Ilari Liukko. UCOT: semiautomatic generation of conceptual models from use case descriptions. In *Proceedings of the IASTED International Conference on Software Engineering, SE '08*, page 171–177, Innsbruck, Austria, 2008. ACTA Press. ACM ID: 1722635.
- [Kof07] L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 121–130. IEEE, October 2007.
- [Kof08] L. Kof. From textual scenarios to message sequence charts: Inclusion of condition generation and actor extraction. In *16th IEEE International Requirements Engineering, 2008. RE '08*, pages 331–332. IEEE, September 2008.
- [Kof10] L. Kof. From requirements documents to system models: A tool for interactive Semi-Automatic translation. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 391–392. IEEE, October 2010.
- [Kon12] Konigi. OmniGraffle sketch stencils. <http://konigi.com/store/product/omnigraffle-sketch-stencils>, April 2012.
- [Kru99] Philippe Kruchten. Use-Case storyboards in the rational unified process. In *Proceedings of the Workshop on Object-Oriented Technology*, page 249–250, London, UK, 1999. Springer-Verlag.
- [Kru04] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, Bosten, MA, USA, 2004.
- [KS07] Margrethe Adde Kjeøy and Gerd Melteig Stalheim. Use cases in practice: A study in the norwegian software industry. June 2007.
- [Kuj05] Sari Kujala. Linking user needs and use Case-Driven requirements engineering. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 113–125. Springer-Verlag, Berlin/Heidelberg, 2005.
- [Lan96] James A. Landay. SILK. In *Conference companion on Human factors in computing systems common ground - CHI '96*, pages 398–399, Vancouver, British Columbia, Canada, 1996.

- [LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. page 5. ACM Press, 2006.
- [Li09] Yang Li. Beyond pinch and flick: Enriching mobile gesture interaction. *Computer*, 42(12):87–89, December 2009.
- [Lie12] Michael Liedtke. Android market checks out, google play moves in. <http://news.yahoo.com/android-market-checks-google-play-moves-180102522.html>, March 2012.
- [LM95] James A Landay and Brad A Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, page 43–50, Denver, Colorado, United States, 1995. ACM Press/Addison-Wesley Publishing Co. ACM ID: 223910.
- [LM01] J.A. Landay and B.A. Myers. Sketching interfaces: toward more human interface design. *Computer*, 34(3):56–64, 2001.
- [LNHL00] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. DENIM: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, page 510–517, New York, NY, USA, 2000. ACM.
- [LSZ94] H. Lichter, M. Schneider-Hufschmidt, and H. Züllighoven. Prototyping in industrial software projects-bridging the gap between theory and practice. *IEEE Transactions on Software Engineering*, 20(11):825–832, November 1994.
- [LTL02] James Lin, Michael Thomsen, and James A Landay. A visual language for sketching large and complex interactive designs. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, page 307–314, Minneapolis, Minnesota, USA, 2002. ACM. ACM ID: 503431.
- [MA93] Kim Halskov Madsen and Peter H Aiken. Experiences using cooperative interactive storyboard prototyping. *Commun. ACM*, 36(6):57–64, June 1993. ACM ID: 163268.
- [Mar02] A. Marcus. Return on investment for usable user-interface design: Examples and statistics. *Aaron Marcus and Associates, Inc. Whitepaper*, 2002.
- [May99] Deborah J. Mayhew. *The usability engineering lifecycle: a practitioner's handbook for user interface design*. Morgan Kaufmann, 1999.

- [MCP⁺06] Michael McCurdy, Christopher Connors, Guy Pyrzak, Bob Kanefsky, and Alonso Vera. Breaking the fidelity barrier: an examination of our current characterization of prototypes and an example of a mixed-fidelity success. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, page 1233–1242, New York, NY, USA, 2006. ACM.
- [Mem09] Thomas Memmel. *User Interface Specification for Interactive Software Systems*. PhD thesis, April 2009.
- [MJ02] Stevan Mrdalj and Vladan Jovanovic. User interface driven system design. *ISSUES IN INFORMATION SYSTEMS*, (Volume III), 2002.
- [MO05] Eduard Metzker and Michael Offergeld. An interdisciplinary approach for successfully integrating Human-Centered design methods into development processes practiced by industrial software development organizations. In Murray Reed Little and Laurence Nigay, editors, *Engineering for Human-Computer Interaction*, volume 2254, pages 19–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [MRJ00] Wendy E Mackay, Anne V Ratzer, and Paul Janecek. Video artifacts for design: bridging the gap between abstraction and detail. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS '00, page 72–82, New York, NY, USA, 2000. ACM. ACM ID: 347666.
- [MYBM91] Allan MacLean, Richard M. Young, Victoria M. E. Bellotti, and Thomas P. Moran. Questions, options, and criteria: elements of design space analysis. *Hum.-Comput. Interact.*, 6(3):201–250, September 1991.
- [ND92] D. G Novick and S. A Douglas. QUID: a quick user-interface design method using prototyping tools. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992*, volume ii, pages 709–718 vol.2. IEEE, January 1992.
- [Nie93] Jakob Nielsen. *Usability Engineering*. Academic Press, Boston, September 1993.
- [NO05] Jerzy Nawrocki and Łukasz Olek. UC workbench – a tool for writing use cases and generating mockups. In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *Lecture Notes in Computer Science*, pages 230–234. Springer Berlin / Heidelberg, 2005. 10.1007/11499053_34.
- [Nor02] D.A. Norman. *The design of everyday things*. Basic Books, New York, NY, USA, 2002.

- [Nun01] Duarte Nuno Jardim Nunes. *Object Modeling for User-Centered Development and User Interface Design: The Wisdom Approach*. PhD thesis, April 2001.
- [Ove06] Doug Overton. 'No fault found' returns cost the mobile industry \$4.5 billion per year. <http://www.wds.co/news/whitepapers/20060717/20060717.asp>, July 2006.
- [PPAH05] Pardha S. Pyla, Manuel A. Pérez-Quiñones, James D. Arthur, and H. Rex Hartson. Ripple: An event driven design representation framework for integrating usability and software engineering life cycles. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 245–265. Springer-Verlag, Berlin/Heidelberg, 2005.
- [PV09] F. Perez and P. Valderas. Allowing End-Users to actively participate within the elicitation of pervasive system requirements through immediate visualization. In *Requirements Engineering Visualization (REV), 2009 Fourth International Workshop on*, pages 31–40, 2009.
- [RB00] A. Ravid and D. M. Berry. A method for extracting and stating software requirements that a user interface prototype contains. *Requirements Engineering*, 5(4):225–241, December 2000.
- [RBAC⁺98] C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté, A. Sutcliffe, N. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois, and P. Heymans. A proposal for a scenario classification framework. *Requirements Engineering*, 3(1):23–47, March 1998.
- [RBP⁺91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen. *Object-oriented modeling and design*. Prentice-Hall, Inc., 1991.
- [Ret94] Marc Rettig. Prototyping for tiny fingers. *Communications of the ACM*, 37(4):21–27, April 1994.
- [Rob05] Dave Roberts. Coping with complexity. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 201–217. Springer-Verlag, Berlin/Heidelberg, 2005.
- [RSI96] Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, January 1996.

-
- [SB09] Vinícius Costa Segura and Simone Diniz Barbosa. UISK: supporting Model-Driven and Sketch-Driven paperless prototyping. In *Proceedings of the 13th International Conference on Human-Computer Interaction. Part I: New Trends*, page 697–705, Berlin, Heidelberg, 2009. Springer-Verlag.
- [Sch96] Kurt Schneider. Prototypes as assets, not toys: why and how to extract knowledge from prototypes. In *Proceedings of the 18th international conference on Software engineering, ICSE '96*, page 522–531, Washington, DC, USA, 1996. IEEE Computer Society.
- [Sch07] Kurt Schneider. Generating fast feedback in requirements elicitation. In *Proceedings of the 13th international working conference on Requirements engineering: foundation for software quality, REFSQ'07*, page 160–174, Berlin, Heidelberg, 2007. Springer-Verlag. ACM ID: 1768916.
- [SDM05] Ahmed Seffah, Michel C. Desmarais, and Eduard Metzker. HCI, usability and software engineering integration: Present and future. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 37–57. Springer-Verlag, Berlin/Heidelberg, 2005.
- [SGD05] Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais. An introduction to Human-Centered software engineering. In Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais, editors, *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle*, volume 8, pages 3–14. Springer-Verlag, Berlin/Heidelberg, 2005.
- [SM04] Ahmed Seffah and Eduard Metzker. The obstacles and myths of usability and software engineering. *Commun. ACM*, 47(12):71–76, December 2004. ACM ID: 1035136.
- [SM10] M. Sutherland and N. Maiden. Storyboarding requirements. *IEEE Software*, 27(6):9–11, December 2010.
- [Sny03] Carolyn Snyder. *Paper prototyping: the fast and easy way to design and refine user interfaces*. Morgan Kaufmann, San Francisco, CA, USA, 2003.
- [SP00] Paulo Silva and Norman W. Paton. UMLi: the unified modeling language for interactive applications. In Andy Evans, Stuart Kent, and Bran Selic, editors, *«UML» 2000 — The Unified Modeling Language*, volume 1939, pages 117–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
- [STG03] Reinhard Sefelin, Manfred Tscheligi, and Verena Giller. Paper prototyping - what is it good for? In *CHI '03 extended abstracts on Human factors*

- in computing systems - CHI '03*, page 778, Ft. Lauderdale, Florida, USA, 2003.
- [Sut03] A. Sutcliffe. Scenario-based requirements engineering. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 320–329, 2003.
- [SY06] S. R Subramanya and B. K Yi. User interfaces for mobile content. *Computer*, 39(4):85–87, April 2006.
- [TB90] Steven D. Tripp and Barbara Bichelmeyer. Rapid prototyping: An alternative instructional design strategy. *Educational Technology Research and Development*, 38(1):31–44, March 1990.
- [TBBS06] Maryam Tohidi, William Buxton, Ronald Baecker, and Abigail Sellen. Getting the right design and the design right. In *Proceedings of the SIGCHI conference on Human Factors in computing systems, CHI '06*, page 1243–1252, New York, NY, USA, 2006. ACM.
- [THA06] Khai N Truong, Gillian R Hayes, and Gregory D Abowd. Storyboarding: an empirical determination of best practices and effective guidelines. In *Proceedings of the 6th conference on Designing Interactive systems, DIS '06*, page 12–21, University Park, PA, USA, 2006. ACM. ACM ID: 1142410.
- [TSLD02] Barbara Tversky, T Stahovic, J Landay, and R Davis. What do sketches say about thinking? In *AAAI Spring Symposium on Sketch Understanding*. AAAI Press, 2002.
- [Ver89] L. Vertelney. Using video to prototype user interfaces. *ACM SIGCHI Bulletin*, 21:57–61, October 1989.
- [VSK96] Robert A. Virzi, Jeffrey L. Sokolov, and Demetrios Karis. Usability problem identification using both low- and high-fidelity prototypes. In *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground, CHI '96*, page 236–243, New York, NY, USA, 1996. ACM.
- [VvLMP04] Hung Tran Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 218–228, 2004.
- [Was96] A. I Wasserman. Toward a discipline of software engineering. *IEEE Software*, 13(6):23–31, November 1996.
- [Wik12] Wikipedia. Storyboard. <http://en.wikipedia.org/wiki/Storyboard>, March 2012.

- [Won92] Yin Yin Wong. Rough and ready prototypes: lessons from graphic design. In *Posters and short talks of the 1992 SIGCHI conference on Human factors in computing systems*, CHI '92, page 83–84, New York, NY, USA, 1992. ACM.
- [WPJH98] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: current practice. *Software, IEEE*, 15(2):34–45, 1998.
- [WTL02] Miriam Walker, Leila Takayama, and James A L. High-Fidelity or Low-Fidelity, paper or computer? choosing attributes when testing web prototypes. *PROC. HUMAN FACTORS AND ERGONOMICS SOCIETY 46TH ANNUAL MEETING*, pages 661—665, 2002.