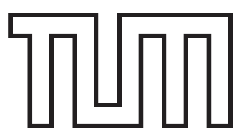


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
LEHRSTUHL FÜR ANGEWANDTE SOFTWARETECHNIK

Demo Engineering: Using Software Theater for Exploratory Projects

Han Xu

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Nassir Navab

Prüfer der Dissertation: 1. Prof. Dr. Bernd Brügge
2. Prof. Dr. Anne Brüggemann-Klein

Die Dissertation wurde am 15.07.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23.11.2020 angenommen.

Abstract

With the growing maturity of new enabling technologies such as smart sensors, AR/VR, wearable computers, IoT, more and more projects of exploratory nature are emerging. In this dissertation, we introduce Software Theater as an alternative way for describing requirements based on screenplays. In particular Software Theater supports the delivery of systems by demonstrating the requirements through theatrical plays. The goal was to find a better way to demonstrate the requirements of innovative systems where the traditional textual description of requirements posed communication problems between developers and customers. Our hypothesis is that Software Theater is more intuitive and engaging than textual descriptions for users to understand and envisage a visionary system. Software Theater is particularly suitable for the co-development and demonstration of visionary concepts, in particular during the analysis, integration testing and delivery of systems that are based on a new user experience paradigm and new technologies. Software Theater was empirically evaluated in 11 projects with 80 students. On a Likert scale, the majority of the participants validated our hypothesis.

Acknowledgements

There are many people who I should thank for their help in the accomplishment of this dissertation. I would like first to thank my supervisor, Prof. Bernd Bruegge. To me, Bernd is not only a professor, an educator, but also a thinker and sometimes a magician who is capable of turning dreams into reality. I have learned a lot from him academically and philosophically. I would also like to express my thanks to my second supervisor, Prof. Anna Brueggemann-Klein for reviewing and guiding me to finish up the dissertation. I am particularly grateful to my industrial supervisors from Siemens, Dr. Naoufel Boulila and Dr. Oliver Creighton, who have, together with Prof. Bernd Bruegge, defined this challenging but interesting research topic. It was a wonderful journey and I really appreciate it that I was trusted and given the opportunity and freedom to explore the vast ocean of knowledge as I desired, which has resulted in the completion of this dissertation as one thing and more importantly the acquisition of intellectual treasure that will benefit me in the rest of my life. I should also give thanks to Siemens-DAAD scholarship for having sponsored my doctoral research. My special thanks are extended to Ms. Rebekka Kammler and Ms. Irmgard Kasperek from DAAD Special Program 522, and Ms. Monika Markl from TU Munich, who had helped me a lot for my arrival and stay in Germany. Other colleagues at Chair of Applied Software Engineering have contributed and helped with my doctoral research in one way or another. I would like to give thanks to Stephan Krusche and Dora Dzvonyar, for having worked together and co-authored papers on Software Theater, to Juan Haladjian, Zardosht Hodaie, Lara Marie Reimer, Florian Bodilee, and Marko Jovanović for useful discussions on using Software Theater, to Nadine Frankenberg for proof-reading the dissertation, to Helmut Naughton, Barbara Reichart, Florian Schneider, Tobias Roehm, Walid Maalej, Dennis Pagano, who have given me valuable advices at doctoral seminars, to Jan Knobloch, Nitesh Narayan, Hoda Naguib, Yang Li, Emitza Guzman, Stefan Nosovic, Rana Alkadhi for having provided me with strength and support during my process of writing, and to Helma Schneider and Uta Weber for your always kind support to me. I appreciate the insight into the film theory provided to me by Chunguang Zhu. Finally, I would like to express my gratitude to my family, especially my wife for her constant understanding, patience and devotion; and to my parents for their support all the time.

Conventions

Font Conventions

- **Bold** is used for terms when they are defined
- *Italic* is used when emphasizing an important word or phrase in its context
- Term is used when emphasizing a particular term with well-defined meaning
- "Double-quoted text" means the text is quoted from a cited publication

Citation Style

In-text citations in this dissertation use the format [<last name of the first author><year of publication>], such as [Bruegge2012] and [Rosson2012]. For citations of books, we will also specify the page number or chapter number to help readers locate the source, such as [Bruegge2010, p.51] or [Cockburn2000, ch.11].

Table of Contents

Abstract ii
Acknowledgements iii
Conventions iv
Table of Contents v
Chapter 1 Introduction 1
1.1 Problem Statement 2
1.2 Research Scope 2
1.3 Structure of the Dissertation 3
Chapter 2 Foundation 4
2.1 Different Models in Software and Requirement Engineering 4
2.1.1 Two domains in software engineering 5
2.1.2 Three worlds of information systems 6
2.1.3 Four models of software systems 9
2.1.4 Informal models and formal models for communication 11
2.1.5 Communication in requirements engineering 12
2.2 Requirements Description in Scenarios, Stories and Use Cases 13
2.2.1 The CREWS's scenario classification framework 13
2.2.2 Use cases and scenarios in software engineering 17
2.2.3 Scenarios in usability engineering 18
2.2.4 Stories in agile development 19
2.2.5 Personas in user interface design 20
2.2.6 Summary: the anatomy of scenarios 22
2.3 Understanding Dual Perspectives of Scenarios 24
2.3.1 Scenarios as artifacts vs. scenario creation as a process 24
2.3.2 Different forms of scenarios for different purposes 25

2.3.3 Benefits of scenarios reviewed from the dual perspectives 25
2.3.4 Summary: the value of this distinction 28

Chapter 3 Demo Engineering Using Software Theater and Tornado Model................. 29

3.1 Challenges with Exploratory Projects.. 30

3.1.1 Constant changes as in every software development................................ 30
3.1.2 Uncertainties from interplay between different levels 30
3.1.3 Inexperience resulting from novelty 31
3.1.4 Summary: how to address these challenges 32

3.2 Demo Engineering 33

3.2.1 Software Theater: scenario-based demonstration 34
3.2.2 Tornado Model: demo-oriented development 37

3.3 The Workflow of Demo Engineering 40

3.3.1 Preparation 40
3.3.2 Implementation... 41
3.3.3 Presentation 44

3.4 Principles of Demo Engineering and Software Theater..................................... 48

3.4.1 Co-development of the user model and the system model 48
3.4.2 Using prototypes with scenarios.. 51
3.4.3 Focus and minimal effort with demo-oriented development 53
3.4.4 Using theatrical techniques for the demonstration................................... 55

3.5 Related Works on Software Theater.. 57

3.5.1 Laurel's Computers as Theatre 57
3.5.2 Mahaux and Maiden's Improvisional Theater 58
3.5.3 Role-playing based on CRC Cards.. 58
3.5.4 Rice and et al.'s Forum Theatre 59

Chapter 4 Software Theater Patterns and Best Practices.. 60

4.1 Heuristics from Theater Theory.. 60

4.1.1 Forms of the performance 60
4.1.2 Story and storytelling of the performance 62
4.2 Software Theater Patterns 66
4.2.1 Dialogue pattern 67
4.2.2 Narrator pattern 69
4.2.3 Monologue pattern 71
4.2.4 Metaphor pattern 72
4.2.5 Conflict pattern 74
4.2.6 Twist pattern 76
4.2.7 Contrast pattern 77
4.2.8 Straightforward pattern 78
4.3 Software Theater Best Practices 79
4.3.1 Setting up additional screens for synchronized illustration 79
4.3.2 Using video as an alternative to theater 79
4.3.3 Illustrating negative sides as well 81
4.3.4 Getting customers involved in creating demo scenarios 81
Chapter 5 Software Theater Example and Evaluation 82
5.1 The Practical Course of iPraktikum 82
5.2 Example: The Zeyes Project 86
5.2.1 Preparation: from visionary scenarios to formalized scenarios 86
5.2.2 Implementation: create the demo backlog and the demo system 88
5.2.3 Presentation: performing the demonstration using Software Theater 91
5.3 Evaluation 93
5.3.1 Evaluation design 93
5.3.2 Evaluation results 94
5.3.3 Threats to validity 98
Chapter 6 Conclusion 99

6.1 Contributions 99
6.2 Future Works 101
6.2.1 Theater script editor 101
6.2.2 Software for creating scenario videos based on virtual worlds 102
Appendix Examples of Theater scripts 103
Bibliography 111

Chapter 1
Introduction

"I think that in the discussion of natural problems we ought to begin not with the Scriptures, but with experiments and demonstrations."

-- Galileo Galilei

In the early days of computer systems, the main focus of software development was the implementation of functionalities based on data structures, algorithms, and communication mechanisms. In those days, computer systems came with simple command-line interfaces (CLI) and were designed primarily for trained vocational users. However, with the development of advanced graphic user interfaces, computer systems were becoming diversified in terms of purpose and usage, and entered people's daily life. This trend has contributed to the emergence of user-centered design [Norman1986] and participatory design [Schuler1993], where the focus of system development was extended from the system requirements of the system to include the usability and user experience of users as well. In addition, to improve the communication among developers and users, Carroll introduced scenarios, which were defined as "narrative description of what people do and experience as they try to make use of computer systems and applications" [Carroll1995]. Since then, scenarios have been used in requirements engineering and usability engineering. For example, Sutcliffe created a *SCenario Requirements Analysis Method (SCRAM)* that combined the scenario-based approach and the model-based approach with an emphasis on dependency analysis based on models [Sutcliffe1998, Sutcliffe2002]. Jarke et al. developed *CREWS (Cooperative Requirements Engineering With Scenarios)* methods and tools [Jarke1999, Maiden1998] and used scenarios as informal representations of system behaviors for requirements elicitation and validation. The usability engineering community used scenarios for describing user-observable

system interactions in a specific context from the user's perspective; in this context, scenarios were viewed as a common language for communication between developers and users [Carroll2000, p.58; Rosson2001, p.23].

1.1 Problem Statement

Scenarios used in 1990s were mostly based on textual description, which limited them in terms of expressiveness. In addition, according to the "three-world" conceptualization introduced by Jarke et al. [Jarke1993], the content of these scenarios was restricted to the description of *system worlds* and *usage worlds*, not addressing the description of *subject worlds*. The usage world can be basically described using UML models such as use cases showing the interaction between actors and the system. The system world describes the internals of the system in the solution domain. The subject world is involved in the interaction of real user accessing physical objects. For example, a visionary application that uses augmented reality glasses to perform quality inspection of parts produced in a factory requires the modeling of interaction of the user with physical objects to perform the inspection task. In this case, the subject world needs to be addressed which cannot be described with text-based scenarios.

1.2 Research Scope

In this dissertation, we describe *Software Theater* [Xu2015, Krusche2018] as an alternative way for representing requirements and performing demonstration based on screenplays like a theater. The goal was to find a better way for innovative systems to demonstrate the requirements where the traditional textual description of the requirements posed communication problems between developers and users. Software Theater is a new way to demonstrate requirements based on scenarios that allow to involve the subject world. Our hypotheses are:

- H1) Software Theater is suitable for presenting future systems in the context of exploratory projects involving the subject world;

- H2) Software Theater is more intuitive and engaging than textual descriptions for users to "see" (i.e., understand and envisage) the future system;
- H3) Software Theater creates more faithfulness and confidence for the evaluation of the future system.

1.3 Structure of the Dissertation

This dissertation is organized in the following chapters.

- Chapter 2 introduces the necessary background knowledge in software engineering and scenario-based design that forms the foundation for understanding our views and opinions throughout the dissertation. Certain concepts introduced in this chapter will be used as an analysis tool in the remainder of this dissertation.
- Chapter 3 first introduces the challenges faced by exploratory projects nowadays and then describes why Demo Engineering and Software Theater could be used to address these challenges. Next, it elaborates on what Demo Engineering and Software Theater are and how to use them. Finally, it describes the principles behind Demo Engineering and Software Theater as well as related works.
- Chapter 4 first describes the heuristics learned from the theater theory and then introduces Software Theater patterns and best practices that we have identified by carrying on real-world projects. These patterns and best practices are helpful for beginner users to get started with Software Theater quickly.
- Chapter 5 illustrates the use of Software Theater in real-world projects by example and empirically evaluates the use of Software Theater. The results are compared with the hypotheses described above.
- Chapter 6 concludes this dissertation and describes directions for future research, such as a tool for authoring theater scripts, and a software platform for creating video-based scenarios in virtual worlds.

Chapter 2
Foundation

"Don't listen to the person who has the answers;
listen to the person who has the questions."
-- Albert Einstein

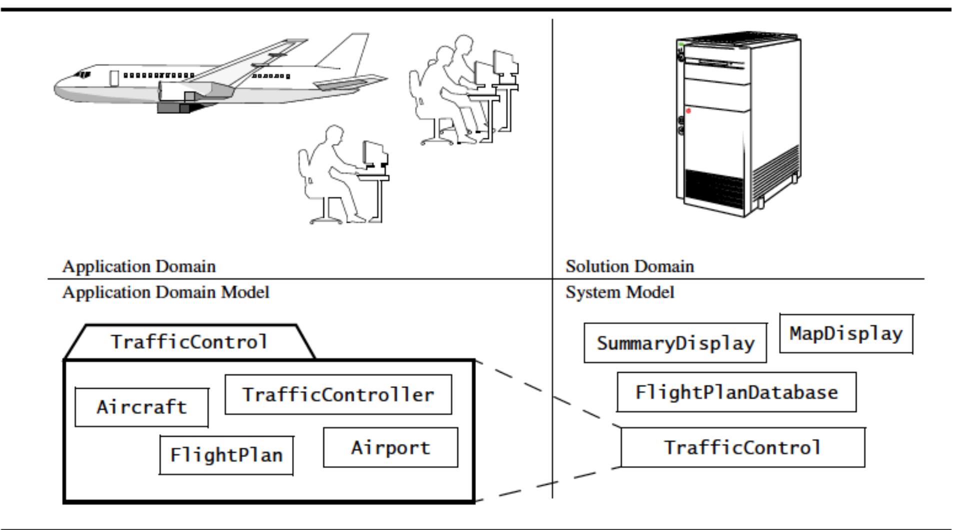
Henry Ford said, "If I had asked people what they wanted, they would have said faster horses." This is a well-known quote that has been frequented cited when talking about new product development. However, it should not be interpreted "why should we get users involved if they cannot tell us that they want a car?" To unravel this puzzle, one needs to recognize the distinction between *application domain* and *solution domain* [Bruegge2010, p.41]; then it becomes naturally to understand the fact that: users are only concerned with their requirements from the application domain, and the developers cannot expect them to possess the knowledge and skills to envision future systems which belong to the solution domain, especially novel systems that they never see before. This is one example that a proper conceptualization is essential in obtaining insight and in-depth understanding. This chapter introduces necessary knowledge that is helpful to understand discussions in the remainder of the dissertation.

2.1 Different Models in Software and Requirement Engineering

Modeling allows us to "focus at any one time on only the relevant details and ignore everything else" [Bruegge2010, p.5]. In this section, we introduce different models that will be used for discussions in the remainder of the dissertation.

2.1.1 Two domains in software engineering

Followed the prologue of this chapter, **application domain** "represents all aspects of the user's problem [including] the physical environment in which the system will run, the users, and their work processes" [Bruegge2010, p.722] and **solution domain** is "the space of all possible systems, [which] focus on system design, object design, and implementation activities." [Bruegge2010, p.746] The purpose of distinguishing application domain and solution domain is the separation of concerns in software engineering practices. The models of software systems are therefore separated into two sections: *application domain models* and *system models* (see Figure 2.1). From software engineering process point of view, system analysis phase deals with the modeling of application domain, and system design phase deals with the modeling of solution domain (see Figure 2.1) [Bruegge2010, p.41].



2.1 The application domain and system domain (from [Bruegge2010, p.42])

While users are concerned with their requirements belonging to the *application domain*, developers are concerned with the technical solutions belonging to the *solution domain* [1]. Users describe the current problems as well as what they what using their knowledge and imagination. Since they have not seen cars before (thus having no concept of *car* in their minds), they will naturally not tell you they want something like a car. Car is a concept belonging to the *solution domain*, and it is the developers' or requirements engineers' responsibility to figure it out (see Figure 2.2).

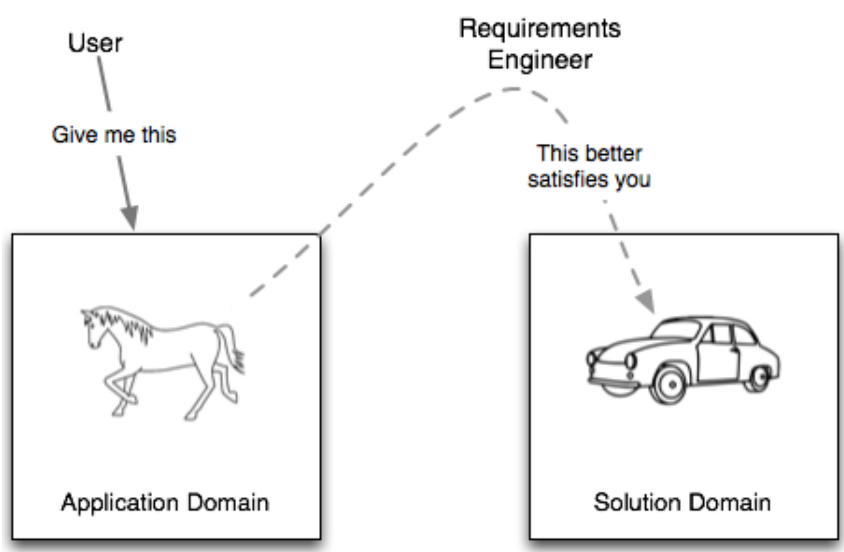


Figure 2.2 Application domain and solution domain

2.1.2 Three worlds of information systems

Jarke et al. proposed *three-world conceptualization* for the modeling of information systems [Jarke1993, Jarke1999]. Unlike the distinction of application domain and solution domain, which is intended for separation of concerns in the software engineering

[1] Developers also need to learn about users' requirements at the same time.

practice, the three-world conceptualization is used to model the *context* or *domain of discourse* of requirements engineering [Jarke1993]. Following this spirit, Jarke et al. defined *subject world*, *system world*, and *usage world* of information systems (see Figure 2.3) [Jarke1992, Jarke1999]:

- **Subject world** is the application domain that should be faithfully represented by the information in the system. The subject world is the physical world where the users "touch and feel".
- **System world** is the solution domain (the system implementation) that manages the information representing physical objects in the subject world and exposes an accessible interface to the user. Normally, operations in the system world could create observable consequences in the subject world.
- **Usage world** represents the production environment (or the context) where the system is situated and manipulated by the users.

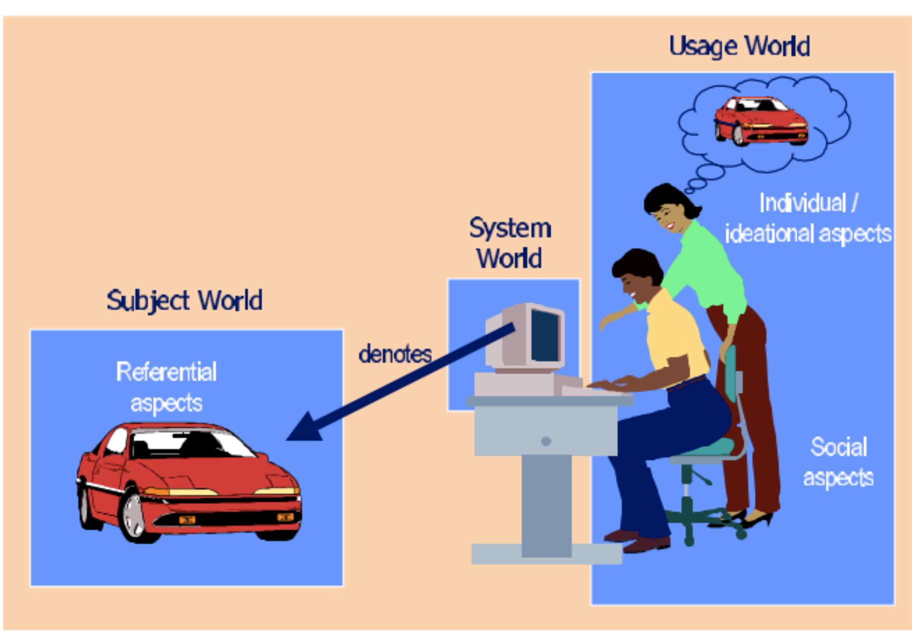


Figure 2.3 Three worlds conceptualization of information systems (from [Jarke1999])

In the remainder of this dissertation, we will use the three-world conceptualization as a reference framework to describe the dimensions that a specific scenario involves. The three worlds make it easier to precisely describe scenarios that require physical objects as the participatory objects (e.g., ubiquitous applications for smart home scenarios) because the interplay among the user, the system, and the physical participatory objects can be specified separately. For example, in a light remote control app, you could switch on and off the light remotely on your mobile phone. When you touch the **ON** button in the app, the light should be powered on. Other examples of scenarios using three worlds are given in Table 2.1.

Table 2.1 Scenarios described in three worlds

Name of scenario	Scenario description		
	Usage World	System World	Subject World
Roaming the login on another device by scanning QR code	1. Use the app on mobile phone to scan the QR code displayed on PC	2. authorize (client id)	3. The same app on PC automatically logs in without needing to enter the username and password
Transfer photos from one mobile phone to another by shaking in the near	1. Select photos on mobile phone A and put it close to mobile phone B. Both phones shake at the same time	2. transfer (connection, file)	3. Mobile phone B receives the photo sent from mobile phone A
Unlock a shared bike by scanning QR code	1. Use the shared-bike app on mobile phone to scan the QR code	2. unlock (bike id)	3. The Bike is unlocked remotely from the Internet
Open doors of a shared car using mobile phone	2. Find the "car" display in the share-car app. 3. Select book it now	4. unlock (car id)	1. Read the plate number of the car near you 5. Doors of the car are opened

2.1.3 Four models of software systems [2]

In the study of human-computer interaction, Norman distinguishes three different conceptual models of a system in user-centered system development: *system image*, *design model*, and *user model* [Norman1986, Norman2013]. The **system image** is what people can derive from the information available, including documentation, instructions, and online information describing the system. The **design model** [3] (or the designer's conceptual model) is the designer's conception of how the system works. The **user model** (or the user's conceptual model) is formed from the system image as well as from the user's prior knowledge and expectations. The user model is continually refined through the user's interaction with the system [Norman1986]. It should be noted that in theory the user model could not be directly constructed from the design model.

Norman's conceptual models were proposed for describing concepts in the domain of user interaction design. In this context, since design models and user models only focus on user-perceivable aspects of the system (i.e., the user interface and interaction), therefore, both user models and design models are about the application domain of the system. However, when it comes to software engineering, the developer's conceptual model should cover not only the user interface, but also technical details relevant to the underlying implementation. To fill this gap, Bernd et al. extended Norman's three models to a *four-model conceptualization* (see Figure 2.4) [Bruegge2012, Xu2013]:

- **System Model** describes the technical details of the underlying structure and behavior of the system implementation. The system model serves as an abstraction of the implementation so that developers could use it as a convention for technical communication and concentrate on technical things.

[2] The account of the four models in this section is revised and adapted from Section 2.5 of [Xu2013].

[3] Note that the design model mentioned here should be distinguished from design models used in object-oriented design.

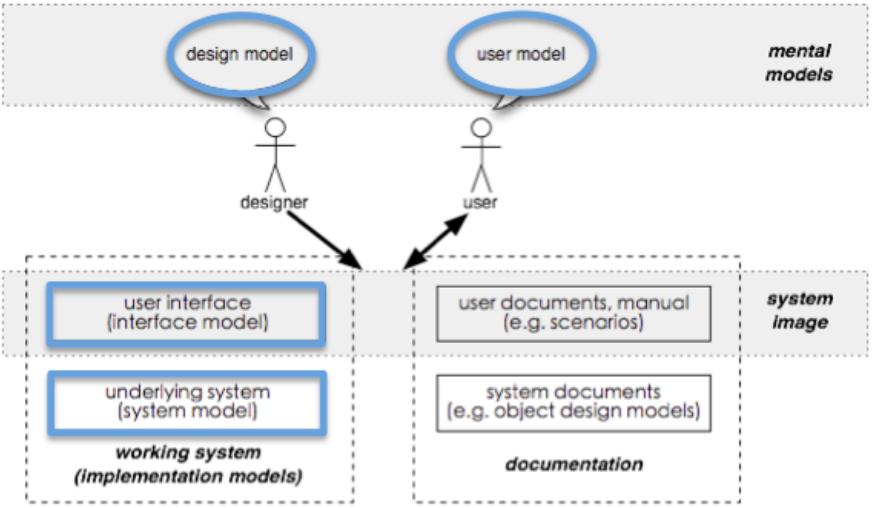


Figure 2.4 Four models of software systems (from [Xu2013])

- **Design Model** is the designer's mental model about how the interaction should happen between the user and the system. Since the user only cares about the user-perceivable behavior of the system, the design model is mainly about the user interface and interaction.

- **User Model** is the user's mental model about how the system should work in specific contexts from the usage point of view.

- **Interface Model**. In order to hide the technical complexity of the system model, the interface model is spin off from the system model, referring to the user-visible aspects of the system [Bruegge2015]. It should be noted that the system model in a broad sense comprises user interface components as well. If not otherwise mentioned in the remainder of this dissertation, when we mention *system model*, it refers to the broad sense of system model.

Both the design model and the user model are mental models, which means they are "conceptual models formed through experience, training, and instruction" [Norman2013]. Ideally, user models should be consistent with design models; however, in practice, we

need much effort to maintain the consistency between design models and user models. For the users to be satisfied with the system implementation, the design models should be consistent with the user models and are faithfully implemented by the system models and interface models. One of the ways to achieve this goal is to get users involved in the iterations of software design and evaluation because it provides an opportunity to verify the design models with the user models, and vice versa. Besides, communication is crucial in guaranteeing the efficacy of user involvement, which will be discussed in more detail in Section 2.1.5.

Unlike the application domain and solution domain in software engineering (as well as the associated application domain models and system models) and the three worlds of information systems, which are objective concepts, four models of software systems are not all objective. While the system model and the interface model, belonging to the solution domain, are objective, the user model and the design model, representing the user's and the designer's mental models about the system respectively, are subjective.

2.1.4 Informal models and formal models for communication

Software systems can be described using different levels of formality. **Informal models**, such as user stories and scenarios, are used to describe how to use the system to achieve specific tasks from the user's point of view [Bruegge2012]. Examples of informal models include sketches [Brajnik2014], paper prototyping [Rettig1994], storyboards [Madsen1993], text-based scenarios [Achour1998] and video-based scenarios [Creighton2006, Xu2013]. **Formal models**, such as UML diagrams, are used to describe components or working mechanisms of a system. Formal models are more accurate and rigorous than informal models, which make them suitable for describing technical aspects of software systems and therefore serve as a technical language among the developers. However, UML models would be a nightmare for users who are technically unsophisticated [Pressman2009, p.51]. As summarized by Bruegge et al. [Bruegge2012],

the informal models and formal models are different in purpose and preciseness (see Table 2.2).

Table 2.2 Comparison of informal model and formal model

	Informal models	**Formal models**
Purpose	Communication with users about requirements	Communication among developers about implementation specification
Preciseness	Possibly incomplete, ambiguous, unverified	Rigorous, unambiguous, consistent

2.1.5 Communication in requirements engineering

Communication plays a vital role in requirements engineering. Cohn stated, "software requirements is a communication problem" [Cohn2004, p.3]. There are two challenges in requirements engineering, according to Pressman [Pressman2009, p.119]: 1) users do not know their requirements or cannot express it precisely; 2) users' requirements change over time. Reaching agreement on the requirements requires the joint effort of developers and users. Leffingwell defined the communication gap between users and developers as the *user and developer syndrome* and summarized common problems that were observed in practice (see Table 2.3) [Leffingwell1999, pp.83-84].

Table 2.3 The user and the developer syndrome (from [Leffingwell1999, p.84])

Problem	**Solution**
Users do not know what they want, or they know what they want but cannot articulate it.	Recognize and appreciate the user as domain expert; try alternative communication and elicitation techniques.
Users think they know what they want until developers give them what they said they wanted.	Provide alternative elicitation techniques earlier: storyboarding, role-playing, throwaway prototypes, and so on.
Analysts think they understand user problems better than users do.	Put the analyst in the user's place. Try role-playing for an hour or a day.
Everybody believes everybody else is politically motivated.	Yes, its part of human nature, so let's get on with the program.

2.2 Requirements Description in Scenarios, Stories and Use Cases

Scenarios, *stories*, and *use cases* are terms that are frequently mentioned in the software engineering literature. Although they have different names, they do share some similarities when looking at concrete examples. Basically, they are all about narrative descriptions of how users interact with systems in order to achieve their goals [Carroll1995]. According to Rolland et al., all these concepts could be loosely considered as a sort of scenario because they all "emphasize some description of the real world" [Rolland1998]. But apart from this commonality, they have subtle differences and are associated with preferred connotations in different contexts. This section will describe the differences between scenarios, use cases, and stories. In particular, we distinguish between the *broad sense of scenario* (or *generic scenarios*) and *narrow sense of scenarios* (or *specific scenarios*). This distinction is meaningful because, on the one hand, it gives us insight into where these concepts share properties in common so that we could reuse experiences gained from one concept for another; and on the other hand, it prevents us from using terms ambiguously in the literature. In the following, we first describe the scenario classification framework proposed by Rolland et al. [Rolland1998], then review and compare specific scenarios based on this framework.

2.2.1 The CREWS's scenario classification framework

In the CREWS project [Jarke1999], Rolland et al. proposed a *classification framework for scenarios* [Rolland1998]. The *scenarios* here refer to the broad sense of scenario, covering text, graphics, image, video, or software prototype depending on the Medium. The framework classifies scenarios from four different views: Form, Content, Purpose and Lifecycle (see Figure 2.5). Each of these views, when applicable, is further divided into different Facets; and each Facet is measured by a set of relevant Attributes, whose possible type of value is defined by the Domains [Rolland1998]. In the remainder of this

dissertation, when the terms Form, Content, and Purpose are used in association with scenarios, they have the meanings described below:

- "The Form view deals with the expression mode of a scenario. Are scenarios formally or informally described, in a static, animated or interactive form? These are the kinds of questions about scenarios which emerge from this view.
- The Content view concerns the kind of knowledge which is expressed in a scenario. Scenarios can, for instance, focus on the description of a system functionality or they can describe a broader view in which the functionality is embedded into a larger business process with various stakeholders and resources bound to it.
- The Purpose view is used to capture the role that a scenario is aiming to play in the RE process. Describing the functionality of a system, exploring design alternatives or explaining drawbacks or inefficiencies of a system are examples of roles that can be assigned to a scenario." [Rolland1998]

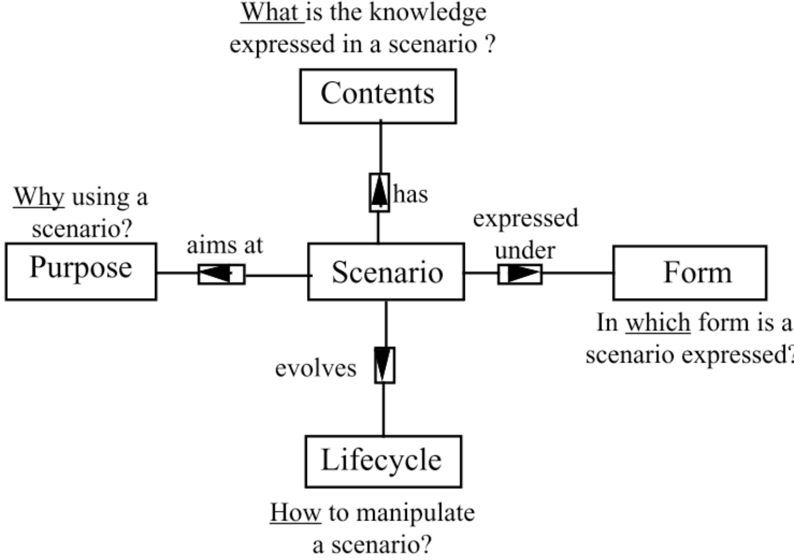


Figure 2.5 Classification Framework for Scenarios (from [Rolland1998])

Table 2.4 summarizes the CREWS' scenario classification framework in terms of Views, Facets, Attributes and Domains.

Table 2.4 Scenario classification framework proposed by the CREWS Project (created according to [Rolland1998])

View	Facet	Attribute	Domain	Description
Form	Description	Medium	SET (ENUM {text, graphics, image, video, software prototype})	Medium used for the description
		Notations	ENUM {formal, semi formal, informal}	The formality level of the notations used for the description
	Presentation	Animation	BOOLEAN	Scenarios can be in animated or static. Animated presentations highlight the expected behavior of the future system in a natural way. Static presentations are in graphics and/or text.
		Interactivity	ENUM {none, hypertext-like, advanced}	Scenarios can have a certain level of interactivity. Hypertext-like interactivity allows the user to follow hypertext links. Advanced interactivity allows the user to trigger different actions and events based on the user's choice.
Content	Abstraction	Instance	BOOLEAN	Instance scenarios or concrete scenarios describe details of individual actors, events and episodes with little or no abstraction.
		Type	BOOLEAN	Type scenarios or abstract scenarios are described in a more abstract way without mentioning particular entity instances. Type scenarios can vary in the level of abstraction.
		Mixed	BOOLEAN	Mixed scenarios contain different levels of abstraction.
	Context	System internal	BOOLEAN	The scenario describes the interaction between components within the system boundary.
		System interaction	BOOLEAN	The scenario treats the system as a black box and only describes the

				interaction between the system with its environment including the actor or other system.
		Organizational context	BOOLEAN	The scenario contains explanation of the application domain knowledge, including knowledge on the stakeholders' motivation, goals, social relationships and responsibilities etc.
		Organizational environment	BOOLEAN	The scenario contains information about the environment where the organization is in, such as team distribution and government regulations etc.
	Argumentation	Positions	BOOLEAN	Descriptions of alternative solutions to a problem
		Arguments	BOOLEAN	Arguments for objecting or supporting a given position.
		Issues	BOOLEAN	Descriptions of problems or conflicts of the solution.
		Decisions	BOOLEAN	Choices of a particular position.
	Coverage	Functional	SET (ENUM {structure, function, behavior})	Most scenario models focus on descriptions of behavioral aspects.
		Intentional	SET (ENUM {goal, problem, responsibility, opportunity, cause, goal dependency})	Intentional models are seldom included in scenario approaches
		Non-functional	SET (ENUM {performance, time constraints, cost constraints, user support, document, examples, backup/recovery, maintainability, flexibility, portability, security/safety, design constraints, error handling})	Only a few scenario models give explicit and extended guidelines about what kind of non-functional requirements on should express, and how to express them.
Purpose	Descriptive		BOOLEAN	Primarily for the purpose of capturing requirements. Also useful to investigate the opportunities for process

				improvements or to investigate the impacts of a new system and therefore for business process re-engineering.
	Exploratory		BOOLEAN	Scenarios used to explore and evaluate possible solutions to support an argumented choice. Helpful in exploring various solutions and selecting the most appropriated one
	Explanation		BOOLEAN	Scenarios that provide detailed illustrations of these situations and their rationale
Lifecycle	Lifespan	Lifespan	ENUM {transient, persistent}	Transient scenarios are thrown after being used. Persistent scenarios exist as long as the documentation of the project they belong to.
	Operation	Capture	ENUM {fromscratch, byreuse}	About the generation of scenarios, from scratch or by reuse.
		Integration	BOOLEAN	Whether or not the scenario is integrated, which means the scenario is combined based on fragmented pieces.
		Refinement	BOOLEAN	Whether or not the scenario is refined, which means the scenario is improved in terms of understandability and reusability without increasing content.
		Expansion	BOOLEAN	Whether or not the scenario is expanded, which means new knowledge is added.
		Deletion	BOOLEAN	Whether or not the scenario is deleted.

2.2.2 Use cases and scenarios in software engineering

Use cases have the root in software engineering and are used for the Purpose of describing user requirements [Jacobson1992; Bruegge2010, p.121]. According to Bruegge et al., "use cases describe the behavior of the system as seen from an actor's point of view. Behavior described by use cases is also called external behavior. A use case describes a function provided by the system as a set of events that yields a visible result

for the actors. Actors initiate a use case to access system functionality. The use case can then initiate other use cases and gather more information from the actors." [Bruegge2010, pp.44-46] As a comparison, a scenario means an instance of use case in software engineering.

In practice, software engineering, typically, we need to create multiple use cases to cover the complete functionality of the system. As Bruegge et al. described: "a **use case** is an abstraction that describes all possible scenarios involving the described functionality. A **scenario** is an instance of a use case describing a concrete set of actions. Scenarios are used as examples for illustrating common cases; their focus is on understandability. Use cases are used to describe all possible cases; their focus is on completeness." [Bruegge2010, p.50] Goldsmith also pointed out that use cases should describe all the possible paths that may happen when an actor uses the system to achieve his goal, including both successful paths and failure paths [Goldsmith2004, p.154].

2.2.3 Scenarios in usability engineering

Unlike use cases, which are mainly used in software engineering, scenarios are used widely in usability engineering as well [Rosson2012]. Carroll defined **scenario** in his *Scenario-Based Design* book as "a narrative description of what people do and experience as they try to make use of computer systems and applications" [Carroll1995]. As a comparison to *scenarios in software engineering* being defined as instances of use cases, scenarios in usability engineering focus on the concrete description of the interaction with the system from the user's perspective [Carroll1995]. Although there is a subtle difference between scenarios (in usability engineering) and use cases (as well as scenarios as instances of use cases), they share one commonality in terms of Content: they all describe sequences of events in terms of input and output of the system. Besides, in terms of Abstraction (refer to Section 2.2.1), use cases are more general than scenarios and "intended to be a complete description of what a system will do. [...] Scenarios focus less on completeness of coverage." [Rosson2001, p.19]

Bruegge et al. from the software engineering community pointed out that, "scenarios cannot (and are not intended to) replace use cases, as they focus on specific instances and concrete events (as opposed to complete and general descriptions)." [Bruegge2010, p.132] Rosson and Carroll proposed a way of integrating use cases and scenarios in usability engineering: "one way to integrate the two methods is to develop use cases as a functional specification of user-system exchanges, and write scenarios that raise and consider the usability implications of these exchanges." [Rosson2001, p.19]

Regarding the difference between scenarios and stories, Cohn from the agile community believed that "interaction design scenarios are much more detailed than user stories, often describing the persona and the context of the system use in great detail. Also, a scenario often describes a broader scope than does a user story." [Cohn2004, p.254] Apart from those mentioned above, according to Carroll et al., there is yet another major difference between use cases and scenarios in terms of Context (refer to Section 2.2.1): use cases do not cover motivation, experience and usability issues, but scenarios do [Carroll1998].

2.2.4 Stories in agile development

User stories were initially used in Extreme Programming (XP) [Beck1999] as a sort of lightweight requirement allowing developers and customers to set up common understanding of user requirements in a more natural way. This is achieved by taking prose text as the Form to describe requirements in the user's everyday language, instead of using structured formats (like use cases), the Form that is usually for describing system functions from the technical point of view. As described by Bruegge et al., "a user story is a single functional requirement formulated by the customer that is realized and integration tested during an iteration. At the end of iteration, a release candidate is produced and demonstrated to the customer." [Bruegge2010, p.471] Cohn explained, "a user story describes functionality that will be valuable to either a user or purchaser of a

system or software. User stories are composed of three aspects: A written description of the story used for planning and as a reminder; Conversations about the story that serve to flesh out the details of the story; Tests that convey and document details and that can be used to determine when a story is complete." [Cohn2004, p.4]

Apart from requirements elicitation, another Purpose of user stories in XP is to drive the project planning [Bruegge2010, p.670]. Unlike use cases, which describe functionalities planned for the whole project, user stories only focus on the functionalities that are to be implemented in a sprint. As Cohn described, "use cases are often permanent artifacts that continue to exist as long as the product is under active development or maintenance. Stories, on the other hand, are not intended to outlive the iteration in which they are added to the software." [Cohn2004, p.139]

2.2.5 Personas in user interface design

Unlike use cases and stories, personas are, by definition, not a subtype of scenarios but are relevant to and often used together with scenarios. Personas were originally used by Cooper in user interface design and gained popularity when personas were used as a key component in his Goal-Directed Design (GDD) methodology [Cooper2004]. The purpose of personas is to envision the personality, motivation, and preferences of potential end users [Rosson2012]. Personas are, in nature, the models of the end users. As Cooper described, "**personas** are detailed, composite user archetypes that distinct groupings of behaviors, attitudes, aptitudes, goals, and motivations observed and identified during the Research phase. Personas serve as the main characters in a narrative, scenario-based approach to design that iteratively generates design concepts in the Framework Definition phase, provides feedback that enforces design coherence and appropriateness in the Refinement phase, and represents a powerful communication tool that helps developers and managers to understand design rationale and to prioritize features based on user needs. In the Modeling phase, designers employ a variety of methodological tools to synthesize, differentiate, and prioritize personas, exploring different types of goals and

mapping personas across ranges of behavior to ensure there are no gaps or duplications." [Cooper2007]

The persona method works because designers could capture the sympathy of the end users by envisioning how the users would do in a specific context. In this sense, personas and scenarios are related concepts, and in practice scenarios are often associated with a role or persona describing the goal and character of a user. A persona describes the profile and behavior of a typical category of user pertaining to the system. As Gudjonsdottir stated, "personas are fictitious characters that represent the needs of the intended users, and scenarios complementing the personas describe how their needs can be met." [Gudjonsdottir2010] Some practitioners also consider personas as a way of communicating user requirements [Long2009].

Although personas and scenarios are not the same types of artifact, persona method and scenario-based design have something in common, as Rosson and Carroll admitted that "clearly persona-centered design are similar in many ways to SBD" [Rosson2012]. Nielsen also pointed out that scenarios are considered the focal point of the entire persona method. [Nielsen2012] To be specific, personas provide a user-centric framework for envisioning scenarios. Instead of creating scenarios in an unorganized way, personas help establish a user-centric framework for developing scenarios. For example, a designer could start from the characteristics of a typical user, then analyze his or her typical daily tasks, and finally work out required functionalities of the system. Personas provide a mechanism for guiding structured brainstorming and keeping the resulting scenarios relevant to the user. On the other hand, personas ensure that designers will think from the user's perspective, which is helpful in filling the gap between design models and user models.

2.2.6 Summary: the anatomy of scenarios

As described above, scenarios have been widely used in software engineering and usability engineering; however, a consensus on the precise definition of scenario has yet to be reached across different communities. As Sutcliffe has pointed out, "the term 'scenario' has been hijacked by too many authors to have any commonly accepted meaning." [Sutcliffe2002, p.121] In our opinion, different researchers viewed scenarios with different focuses.

- Nielsen highlighted the easy-to-create feature of scenarios and views scenario as "an especially cheap kind of prototype" that supports only limited number of features on the user-interface level [Nielsen1993, p.94].
- Rolland et al. emphasized the narrative nature of scenarios and categorize all the approaches that use examples, scenes, and narrative description of contexts, mock-ups, and prototypes as scenario-based approaches because "these approaches emphasize some description of the real world." [Rolland1998]
- Carroll et al. stressed the user-centric property of scenarios. As Carroll described, "scenarios are stories—stories about people and their activities." [Carroll2000, p.46] Rosson and Carroll et al. described, "scenarios are stories about use, they maintain a central focus on use as the goal of design." [Rosson2004]

In this dissertation, we promote the distinction between scenarios in a broad sense and scenarios in a narrow sense. While authors tend to use *scenarios in a narrow sense* in their own ways, *scenarios in a broad sense* is a umbrella concept that encompasses not only *scenarios in a narrow sense* but also similar concepts like *use cases* and *stories* (see Figure 2.6)[4]. A possible definition for the broad sense of scenario is "the description of the user's interaction with the system in a specific context from the user's perspective"

---

[4] Since in the context of software engineering, the scenario is an instance of use case (see Section 2.2.2), that is why there is an arrowed line from "Scenario in SE" to "Use Case".

[Xu2013]. In the remainder of this dissertation, if not otherwise specified, when we mention 'scenario', we refer to this broad sense of scenario defined here.

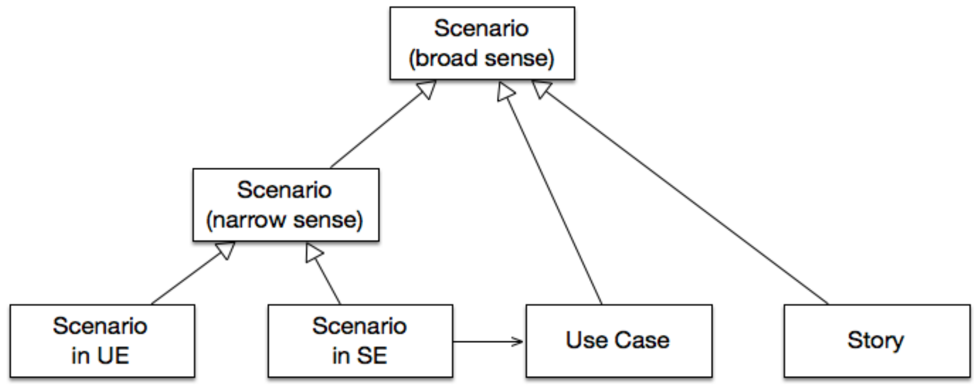


Figure 2.6 Scenarios, stories and use cases

Table 2.4 summarizes the difference of scenarios in software engineering (SE), scenarios in usability engineering (UE), stories, and use cases according to the CREWS classification framework described in Section 2.2.1.

Table 2.4 Comparing scenarios, use case and story

CREWS Attributes	Use cases in SE	Scenarios in SE	Scenarios in UE	Stories in Agile
Form.Description.[Medium]	Text	Text	Text	Text
Form.Description.[Notations]	Semi-formal	Informal	Informal	Informal
Content.Abstraction	Type	Instance	Mixed	Mixed
Content.Context. [System interaction]	True	True	True	True
Content.Context. [Organizational context]	False	False	True	True

2.3 Understanding Dual Perspectives of Scenarios

In this section, we extend the existing understanding of scenarios and promote the distinction of two perspectives of scenarios: *scenarios as artifacts* and *scenario creation as a process*, which are what we call *dual perspectives of scenarios*.

2.3.1 Scenarios as artifacts vs. scenario creation as a process

Whether user stories should take structured or unstructured format has been around in the agile community for a while. According to the description of user stories in Section 2.2, user stories refer to a subclass of scenarios with unstructured prose-text format; thus, the question mentioned above could be reformulated as "whether scenarios should be structured or unstructured?" or further extended as "which Form and level of Abstraction of scenarios should be used for which Purpose?" In this section, we start from this specific question, then take a step further and promote the distinction of dual perspectives of scenarios: *scenarios as artifacts* and *scenario creation as a process*.

Scenarios as artifacts refer to the static perspective of scenarios. As artifacts, scenarios describe the design and usage of the system, which could be used for communication with the stakeholders. Besides, scenarios as artifacts enable developers to conduct design evaluation and rationale analysis with stakeholders. Scenarios for this Purpose should be concise and easy-to-read, and therefore a structured format is appropriate.

Scenario creation as a process refers to the dynamic perspective of scenarios. Scenario creation is a process of idea exploration, where ideas come to mind and go through a quick screening process. This process starts with a goal that guides the creation of scenarios and ends with the scenarios being created as artifacts. This perspective of scenarios highlights the idea generation nature of scenarios. Scenarios for this Purpose provide a context for developers to immerse themselves and generate ideas; therefore, an unstructured format without a predefined framework is appropriate.

2.3.2 Different forms of scenarios for different purposes

Jarke et al. recognized that "the degree of formality required depends on the purpose of scenarios and on the intended audience" [Jarke1998]. Starting from this argument, we open up further discussion below. Scenario creation is a process of brainstorming and thought experiment so that developers could "dry-run" the design ideas using scenarios. For this **Purpose**, scenarios should take a **Form** that facilitates idea generation, possibility exploration, and imagination stimulation without setting any frame that might limit the imagination. Therefore, a prose-text format like user stories is appropriate in this context. Descriptive scenarios, which serve as the artifacts for the **Purpose** of specification and communication, should be concise, clear, and unambiguous; therefore, a structured format such as formalized scenarios and use cases fit best.

2.3.3 Benefits of scenarios reviewed from the dual perspectives

The distinction between scenarios as artifacts and scenario creation as a process brings us in-depth insight into the nature of scenarios. In this section, we recap the benefits of scenarios from the dual perspective of scenarios, respectively.

Benefits from the static perspective of scenarios (as artifacts):

- *Easiness to understand.* Regardless of the **Medium** of a scenario (text, picture, or video), its **Content** is always concrete and story-like. In software engineering, scenarios serve as a common language among stakeholders. Compared to semi-formal models such as UML diagrams, scenarios, as an informal model, are easier to understand for stakeholders without technical background [Pressman2009, p.51].
- *Removal of ambiguity and assumptions.* The concreteness of scenarios helps to clarify implicit assumptions and unforeseen situations in the design. For example, assumed conditions would be subconsciously verified when you read concrete examples of the usage.

* *Better communication and better stakeholder involvement.* As mentioned in Section 2.1.3, stakeholder involvement relies on stakeholder communications. The concrete and story-like description of the system usage "help[s] in bridging the gap between the various stakeholders and the requirements engineers" [Pohl1997]. Haumer et al. also confirmed, "scenarios are proposed as an ideal means to support the definition of the current-state model and to drive the change definition, i.e., to achieve better stakeholder involvement" [Haumer1998]. Besides, "scenarios, because of their concrete and story-like quality, widen the possibility of discussion and negotiation among designers and other stakeholders." [Jiang2010]

* *Basis for Rationale Analysis.* Carroll pointed out that design evaluation and rationale analysis could be conducted based on scenarios. [Carroll2000, p.125] By studying different solutions and identifying positive and negative points, stakeholders can reason about the situation and provide feedbacks about the design. This process also helps to identify new scenarios or update existing scenarios.

* *Cheap form for quick design evaluation and feedback.* In Section 2.1.3, we have mentioned that design evaluation is an important activity during stakeholder involvement. Compared to executable systems, scenarios are cheap to create and easy to modify, which makes it fit for rapid iterations in the new product development process. When new ideas pop up, developers could quickly create scenarios and verify the new design with the stakeholders; after receiving the feedback from the stakeholders, the developers could easily modify the scenarios and verify with the stakeholders once again.

Benefits from the dynamic perspective of scenarios (scenario creation)

* *Situation deduction and design brainstorming.* In its most general sense, the scenario creation process is a situation development deduction process (such as chessboard reasoning). In strategy management, this property of scenarios is employed to form the basis for scenario-based decision making, where scenarios are used as a tool to make situation deduction [Jarke1998]. In software engineering,

scenarios often take easy-to-create Forms such as use cases or user stories, which allows "dry-runs" of the design in a rapid way. As Goodwin described, "scenarios are the stories that drive design decisions" [Goodwin2011a].

- *Stimulate imagination and idea generation.* Scenarios provide a tool for exploring possibilities in requirements and usability engineering. As Jarke pointed out, "the main purpose of developing scenarios is to stimulate thinking about possible occurrences, assumptions relating these occurrences, possible opportunities and risks, and courses of action." [Jarke1998] Go and Carroll also described, "[scenario-based design] provides a good brainstorming tool for planning and allows the stakeholders to consider alternative choices in decision-making" [Go2004].
- *Context supports opportunity and problem identification.* By creating concrete scenarios in specific contexts, potential problems and opportunities could be better identified. Weidenhaupt et al. described, "scenarios present possible ways to use a system to accomplish some desired function." [Weidenhaupt1998] Benyon and Macaulay also agreed, "scenarios raise design questions, suggest design solutions and aid communication within the team" [Benyon2002].
- *Focus enforcement.* When designing a system, developers could be distracted by other designing constraints. Scenarios set up a context for the design and help developers to focus on the relevant aspects of the system that are important to the users, instead of being drifting into technical details. As Carroll described, "sharing and developing scenarios helps to control the uncertainties of design work, while sharpening and strengthening design goals" [Carroll2000, p.55].

2.3.4 Summary: the value of this distinction

Distinguishing between scenarios as artifacts and scenario creation as a process brings us insight into the dual perspectives of scenarios. This insight leads us to review the benefits of scenarios from two perspectives, and provides guidance on "using different **Forms** of scenarios for different **Purposes**", which in turns answers the question "should user stories be structured or unstructured". Besides, Rolland et al. proposed a classification framework for scenarios, but they did not mention the *process* perspective of scenarios; they admitted, "the process dimension of scenarios is seldom considered in the literature." [Rolland1998] The distinction we made in this section highlighted the process perspective of scenarios. We fill this gap because this dissertation emphasizes the importance of the process perspective of scenarios.

Chapter 3
Demo Engineering Using Software Theater and Tornado Model

"Words are but wind; but seeing is believing."
-- Thomas Fuller

Different software engineering approaches have different focuses and different designed purposes. Object-oriented software engineering organizes software engineering activities around building object-oriented models based on user requirements and then resorting to coding and testing. Software systems built with this approach features more flexibility and extendibility as supported by encapsulation, abstraction, inheritance, and polymorphism of object-oriented programming, which makes it suitable for implementing systems like large-scale enterprise information management systems. Scenario-based software engineering organizes software engineering activities around creating scenarios, which focuses more on the development of user requirements and is suitable for user-centered system development.

In this chapter, we describe *Demo Engineering*, which focuses on developing demo systems, creating theater scripts, and presenting demo systems in a theatrical way. Demo Engineering employs *Software Theater* [Xu2015, Krusche2018] for the demonstration and T*ornado Model* [Bruegge2012] for the implementation of demo systems. Software Theater and Tornado Model form the basis for Demo Engineering: they deal with different aspects of demonstration and complement each other in the whole process. Demo Engineering is suitable for the co-development and integration testing of new concepts, new user experience, and new technologies, which makes it a solution to the challenges posed by exploratory projects. In the rest of this chapter, we

first describe the challenges faced and then introduce the concepts of Software Theater and Tornado Model as well as the relevant foundations.

3.1 Challenges with Exploratory Projects

Exploratory projects mentioned in this dissertation mean projects with the goal of creating innovative systems by trying out new concepts, new user experience, and new enabling technologies. These projects are *exploratory* because they are to explore new possibilities rather than focusing on implementation using off-the-shelf technologies. These exploratory projects bring not only novelty but also uncertainties, which are the "two sides of the same coin". In the following of this section, we analyze the software engineering challenges faced by exploratory projects.

3.1.1 Constant changes as in every software development

Change is a common phenomenon in software development and is considered as constant and inevitable [Bruegge2010]. However, certain changes, despite being unavoidable, could be made to happen earlier by speeding up the iteration cycle. For example, some changes occur due to external reasons such as insufficient communication among the stakeholders, inaccurate requirement specification or inappropriate handling of user feedback, etc. The IKIWISI (I'll know it when I see it) issue [Boehm2000, Cao2008] is one of the examples of this category. In our opinion, the impact of these changes could be mitigated by embracing the changes proactively (e.g., facilitating continuous user involvement and rapid iteration).

3.1.2 Uncertainties from interplay between different levels

Exploratory projects need to explore and evaluate innovative ideas. Innovation at different levels brings different degrees of competitiveness. Technical innovation alone is hard to gain traction on the market quickly because it is about the *system world* and not directly tangible to the users; therefore, it takes some time for the users to perceive the

technical benefit underlying the user interface. Concept innovation and user experience innovation based on off-the-shelf technologies have direct visibility to the users but are easier to be followed by other competitors due to the lack of technical barriers. To seek greater competitiveness on the market, one of the strategies is to promote collaboration of innovations at different levels, that is, to try out new possibilities at the levels of concept, user experience, and technology in the parallel and to find the "best" combination. However, this will pose challenges for software engineering because from development process point of view exploratory projects are not like traditional projects that follow a process starting from requirements to implementation; as a comparison, the project team could be requested to look for applicable concepts or killer applications for a specified technology or to renovate user experience by employing emerging technologies. Besides, tryouts at one level may lead to design changes at another level. For example, we are developing a system that relies on imaging sensors for object recognition, and in order to obtain improved performance in one parameter, we need to replace the imaging sensors from one model to another model. However, as every benefit comes with a price, by doing so, we would compromise the performance in another parameter, which is required by a usable design. This kind of interplay between different levels is making the development process of exploratory projects more challenging than traditional ones. As Jarke et al. described, "in these innovation-driven settings, requirements become part of both the business solution and the system solution, and they constantly bridge new solutions to organizational and societal problems … revisiting requirements as implementation progresses and emphasizes the dynamics and intertwining of these activities." [Jarke2011]

3.1.3 Inexperience resulting from novelty

Unlike traditional software systems, exploratory projects today are based on emerging technologies (e.g., deep learning and smart sensors) that enable new concepts (e.g., smart home and sensor-based sports training) and new paradigms of user interaction (e.g., voice recognition and eye control). Some of these innovative systems do not work alone but also need to interact with the surrounding environment. The usability of user

interaction, in these cases, is not only a matter of user interface design but also relies on the use of enabling technologies. For example, let us assume that we are developing an application that employs AR glasses to perform quality inspection of parts. This application requires interaction with physical objects (i.e., that parts produced in a factory) to perform its function (i.e., to check if the parts have any flaw) using a novel interface device (i.e., AR glasses). However, the interaction paradigm of this system goes beyond traditional "keyboard-mouse-display" fashion and is cognitively unfamiliar to the users; to fill the gap, we need to employ scenarios whose Forms are more intuitive and tangible than traditional text-based scenarios. Besides, according to the "three-world" conceptualization (see Section 2.1.2) [Jarke1993], the Content of scenarios in this example involves not only the *system world* and *usage world* but also the *subject world*. In a nutshell, both Content and Forms of scenarios need to be extended to cater for novel systems of this kind.

3.1.4 Summary: how to address these challenges

This section describes how to address the challenges mentioned above. In the development of traditional systems, text-based scenarios are mainly used to represent the design and usage of the system for the purpose of illustration and evaluation. When it comes to exploratory projects, we also need a representation of the system, but with further demands: in terms of Content, it should cover both the *system world* and the *usage world*; in terms of the Form, it should be intuitive enough for the users to "touch and feel" the new system and faithful enough to support reliable evaluation of the system; besides, it should be easy and cost-efficient to create. To summarize, in order to perform a cost-efficient but reliable illustration and evaluation of systems developed in exploratory projects, we need to satisfy the following demands:

- D1: We need a representation of the future system that could represent the current design so that innovative tryouts at different levels (concept, UX, and technology) could be faithfully reflected while keeping the implementation cost minimal.

- D2: We need a way to illustrate the future system in a lifelike setting so that the applicability of new concepts, the usability of new user experience, and the feasibility of new technologies could be evaluated reliably.
- D3: We need to get the environment involved when we are illustrating systems like ubiquitous computing and context-aware computing, and the status of all the three worlds (system world, usage world, and subject world) should be illustrated in the parallel to enable the users to understand the interplay among the user, the system and the environment.

3.2 Demo Engineering

Although different software development approaches will all deliver the software systems implemented in certain programming languages, they follow different processes, have different focuses, and as a result fit for different purposes. Object-oriented software engineering focuses on creating object-oriented models, which will be translated to source code in certain object-oriented programming language. Software systems built with this approach features more flexibility and extendibility coming from high cohesion and low coupling. Scenario-based software engineering focuses on creating scenarios that represent the future systems from the user's point view. This approach highlights the importance of user involvement and is suitable for user-centered software development.

Demo Engineering focuses on working with demos: delivering demo systems using *Tornado Model* [Bruegge2012] and presenting demonstrations using *Software Theater* [Xu2015, Krusche2018]. Instead of seeking decoupling and flexibility of software systems as in object-oriented software engineering, the goal of Demo Engineering is to create *demo systems* for demonstrating the applicability of new concepts, the usability of new user experience, and the feasibility of new technologies. It should be noted that Demo Engineering is not going for the implementation of full-fledged systems. The tenet is to create demo systems (similar to proofs of concept) that faithfully represent innovative

tryouts in the current design while keeping the implementation cost minimal. It is also not a goal to achieve decoupling and flexibility, which actually might not be possible for exploratory projects that try to make extreme use of available technologies because the deep co-design of concept, user experience, and technologies would result in tightly coupled hardware/software systems. In this dissertation, we describe Demo Engineering using Software Theater and Tornado Model as an answer to the challenges faced by exploratory projects (as described in Section 3.1). This section introduces the concept of Software Theater and Tornado Model, and the next section describes the workflow of Demo Engineering.

3.2.1 Software Theater: scenario-based demonstration

Theater, according to Wikipedia, "[has] been an important part of human culture for more than 2500 years and evolved a wide range of different theories and practices" [LouisAlain2019]. As per the definition of *theater* from Marvin Carlson, an American theatrologist, "***theater*** is a collaborative form of fine art that uses live performers, typically actors or actresses, to present the experience of a real or imagined event before a live audience in a specific place, often a stage. The performers may communicate this experience to the audience through combinations of gestures, speech, song, music, and dance. Elements of art, such as painted scenery and stagecraft such as lighting are used to enhance the physicality, presence and immediacy of the experience." [Carlson1986]

Similar to the traditional theater that presents a dramatic story on the stage, the usage of software systems could also be presented with the means of theatric elements [Xu2015, Krusche2018]. **Software Theater** is a way to present demo systems based on screenplays using theatrical techniques, including props, equipment, and performance skills. [Krusche2018]

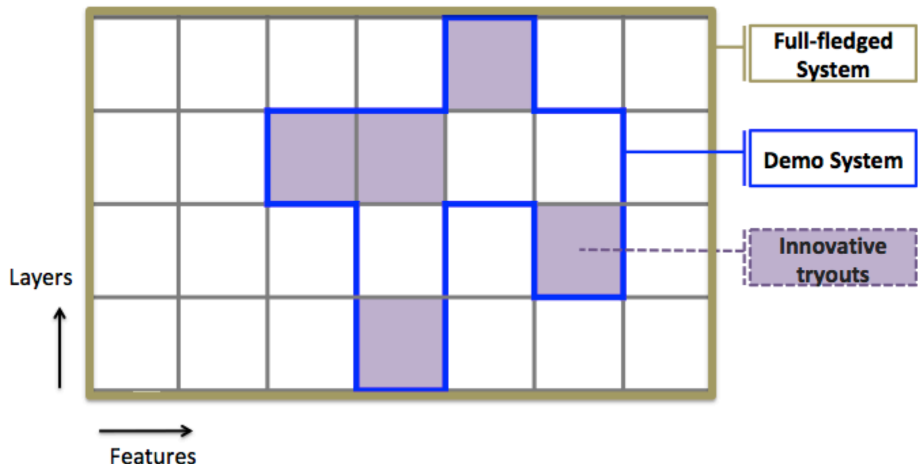


Figure 3.1 The demo system as a subset of the full-fledged system

Software theater, as its name suggests, can be technically explained from two aspects: the *software* aspect and the *theater* aspect.

- The *software* aspect of Software Theater refers to the representations of future systems that are ready for the demonstration. In Demo Engineering, we use *demo systems* to represent and illustrate future systems. A **demo system** is a subset of the full-fledged future system that only represents the innovative tryouts (either concepts, user experience, or technologies) and relevant participating components necessary for the demonstration and evaluation (see Figure 3.1). A demo system is made up of demo components across features and layers. The Forms of demo components are either real implementations (e.g., for evaluating the feasibility of the new technologies) or mock implementations (e.g., for participating components).
- The *theater* aspect is about performing the demonstration on the "stage" [5]. Like in a theater, such a *stage* provides the "real" setup and props that promote "real" experience from the audience and, at the same time, support the evaluation of the

[5] The "stage" is quoted here because it does not require a real stage. It could be a meeting room, a lecture hall, where is arranged like a stage with theatric setup and props.

innovative tryouts. Software Theater creates a lifelike environment and atmosphere, where the actors illustrate the demo system in a theatrical way, highlighting how to use the future system to solve problems and make a difference in everyday life. Similar to illustrating prototypes based on predefined scenarios (see Section 3.4.2), Software Theater presents demo systems based on *theater scripts* (equivalent to *screenplays* in a theater play). The **theater scripts** describe the event flow of the performance as well as relevant details on the participating actors and the props. **Props** are essential for the performance of demonstration. According to Reid, "Props contribute a link between actor and environment. All objects which the actors handle are classified as props...They are an intrinsic part of the action and are not to be confused with dressing the set by placing objects on it for purely visual effect." [Reid1997]

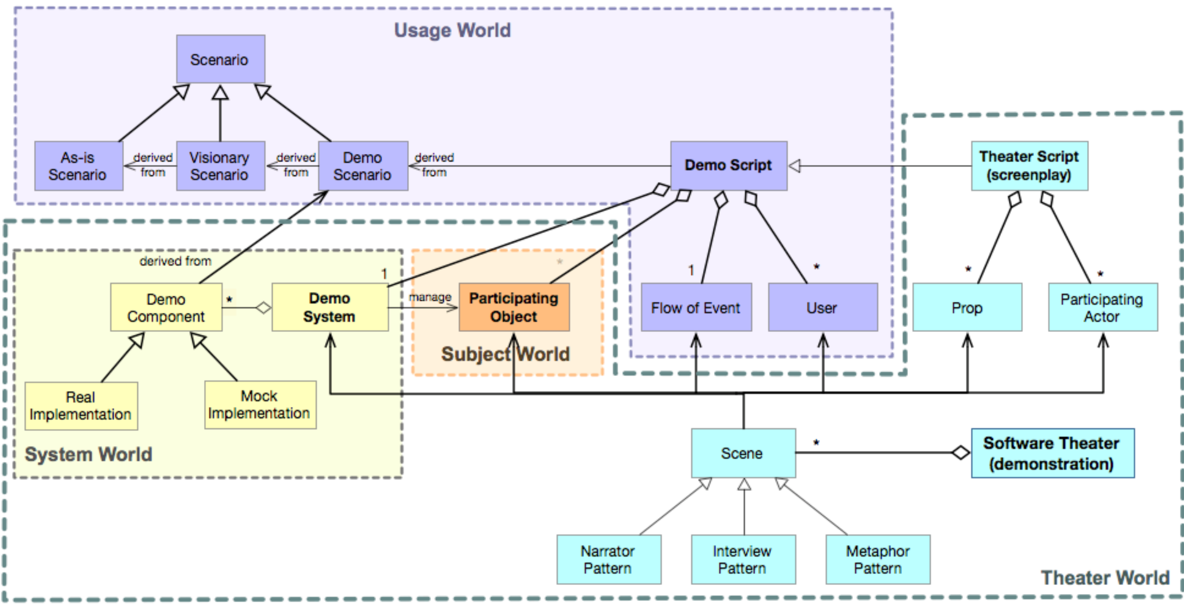


Figure 3.2 The Theater World is another world in addition to system world, usage world, and subject world by focusing on scene, props, and actors.

According to the three-world conceptualization (see Section 2.1.2), the *system world* includes the computer system that manages the state and behavior of participating objects ("subjects") sitting in the *subject world*; and the *usage world* comprises the users as well as the usage of the system (represented in scenarios). With the introduction of *Software Theater*, the three-world conceptualization could be extended to four worlds, with an additional world: *theater world,* which complements the other three worlds (see Figure 3.2). The **theater world** is consist of all the objects needed for performing Software Theater performances, in particular, the demo system from the system world, the participating objects from the subject world, and the demo script from the usage world, etc. In addition, the theater world involves the scenes, props, and real actors.

3.2.2 Tornado Model: demo-oriented development

The **Tornado Model** is a demo-oriented development that aims to deliver systems by providing a series of demos to be demonstrated to stakeholders for evaluation and feedback [Bruegge2012]. These demonstrations can be considered as the "touch point" in a tornado, where the stakeholders could "feel and see" the future system (or part of it) and provide feedback. After the demonstration of "touch point" and the corresponding feedback, the system evolves to the next iteration of demo.

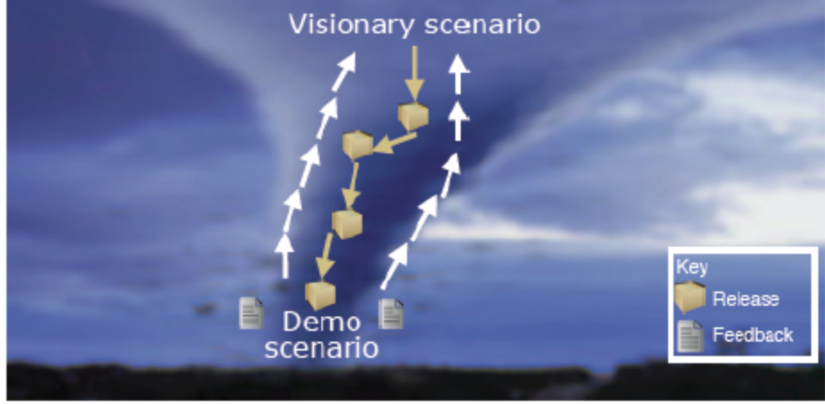


Figure 3.3 Tornado Model provides a process that starts with a visionary scenario and then funnels down to a demo scenario (picture taken from [Bruegge2012])

Figure 3.4 shows a typical project lifecycle following the Tornado Model. The process starts with a visionary scenario and then transforms into a demo scenario. *Visionary scenarios* represent design ideas of the future system and are used for envisioning requirements. In practice, it often requires several rounds of iterations to reach a "stable" version and could have rollbacks. As the main purpose in this phase is to explore different alternatives in the application domain, low-fidelity prototypes are sufficient; therefore, visionary scenarios are usually taking the **Form** of textual descriptions or sketches. *Demo scenarios* are derived from visionary scenarios that are selected for the demonstration (see Figure 3.2). In terms of **Content**, demo scenarios are a refined and normalized version of selected visionary scenarios, illustrating how problems are addressed using the future system from the user's perspective. Demo scenarios consist of a set of demo components, which are are either real implementation (when the focus is the feasibility of new technologies) or mock implementation (when the focus is the applicability of new concepts or usability of new user experience), or a mix of them. Before demonstrating the demo system in Software Theater, demo scenarios need to be converted to *theater scripts* (see Figure 3.2).

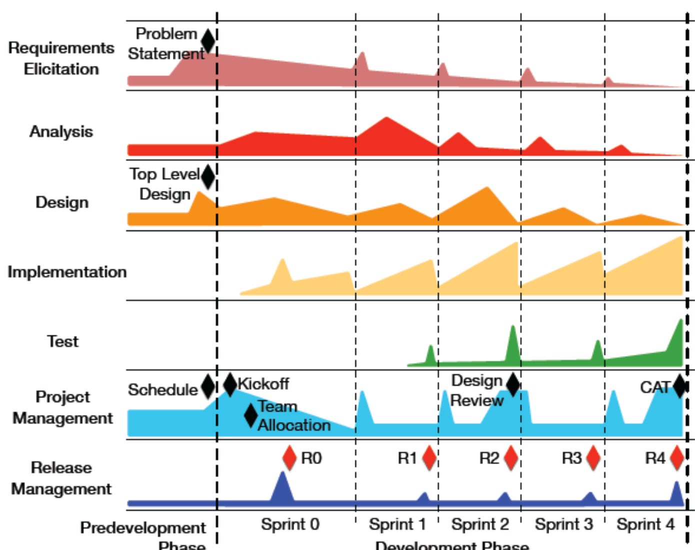


Figure 3.4 A typical project lifecycle following Tornado Model (from [Bruegge2012])

The Tornado Model is also an evolutionary software design process that highlights the importance of the interplay between *user models* and *system models* (see Section 2.1.3). The initial version of the design is depicted using low-fidelity prototypes (for example, a sketchy user interface created with Balsamiq) (see Figure 3.5, left). Low-fidelity prototypes are used in the early phases to get user feedback about the user interface as early as possible in the design process, enabling the users to explore possible design alternatives or to reformulate the requirements. In the middle of the project, as only promising alternatives are left, medium-fidelity prototypes (for example, software mockups as shown in Figure 3.5, middle) are used for a more reliable and tangible evaluation of the design. At the end of the project, the combined design of concept, user experience, and technology that finally wins out is implemented and delivered (as shown in Figure 3.5, right).

Figure 3.5 Evolution of user interface, from low-fidelity to high-fidelity (from [Bruegge2012])

3.3 The Workflow of Demo Engineering [6]

Exploratory projects of different characteristics could use Software Theater and Tornado Model in different ways to cater for their specific situations, but the key ideas should stay the same. Figure 3.6 depicts a reference workflow for Demo Engineering using Software Theater with Tornado Model, which is categorized into three major activities, divided by UML swim lanes: *preparation*, *implementation*, and *presentation*. The first two (preparation of formalized scenarios and implementation of the demo system) is based on Tornado Model, and the last one (presentation of the demo system) is conducted according to Software Theater. We will describe each of these activities in the following sections.

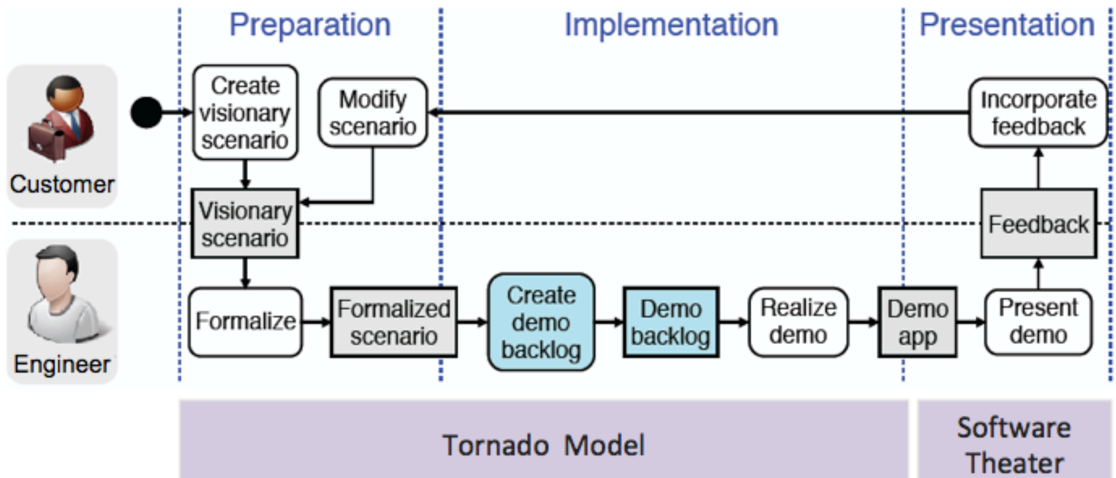


Figure 3.6 Demo Engineering: the reference workflow (adapted from [Krusche2018])

3.3.1 Preparation

First, the project team receives, among others, *as-is scenarios* from the customer (and/or the user, same below) that serve as the problem statement. Then the project team works

[6] This workflow was first described in Section 3 of [Krusche2018] and is refined and adapted in this section.

out *visionary scenarios* together with the customer. As described in Section 2.3.4, different **Forms** of scenarios should be employed for different **Purposes**. In order to stimulate imaginations as much as possible, user stories (refer to Section 2.2.4) are suggested as the format for writing visionary scenarios. It might take several iterations for the project team and the customer to reach an agreement on the visionary scenarios.

Next, visionary scenarios are turned into formalized format (like use cases). A **formalized scenario** "describes the same **Content** as the visionary scenario, but in a structured way" [Krusche2018], which usually comprises six components:

- Scenario name
- Participating actors
- Flow of events
- Entry conditions
- Exit conditions
- Quality requirements

This step is necessary because organizing scenarios in a structured way (formalization) "helps to identify areas of ambiguity as well as inconsistencies and omissions in a requirements specification" [Bruegge2010, p.174]. Besides, it forms the basis for the next steps toward demonstration because structured scenarios help to analyze the event of flow and to identify the actors and props required in the theater script. However, it should be noted that, in practice, the step of creating formalized scenarios could be omitted sometimes, for example, when the scenarios are familiar enough to the project team.

3.3.2 Implementation

The next step is to implement the demo system. As mentioned in Section 3.2, a demo system is a subset of the full-fledged future system, which includes only demo

components that are valuable and meaningful for the live demonstration. The demo system should cover those innovative tryouts reflecting new concepts, new user experience, or new technologies (see Figure 3.1). The scope of the implementation is defined by the *demo backlog*, a list of prioritized tasks for implementation. The demo backlog is created based on demo scenarios, a subset of visionary scenarios that are selected for demonstration; at the same time, demo scenarios rely on the demo backlog to include necessary demo components for performing the demonstration. As depicted in Figure 3.7, to create the demo backlog, it starts from writing demo scenarios. The initial version of demo scenarios is usually derived from formalized scenarios created from the previous step. Then it continues to identify demo components that are necessary for the demonstration. These demo components are to be implemented by the project team. It should be noted that it is normal if certain demo components cannot be implemented as expected in the end, for example, when it turns out to be not technically feasible or unable to meet the desired specification. In this case, the relevant demo scenario needs to be modified by removing the parts that are affected by the absence of the component.

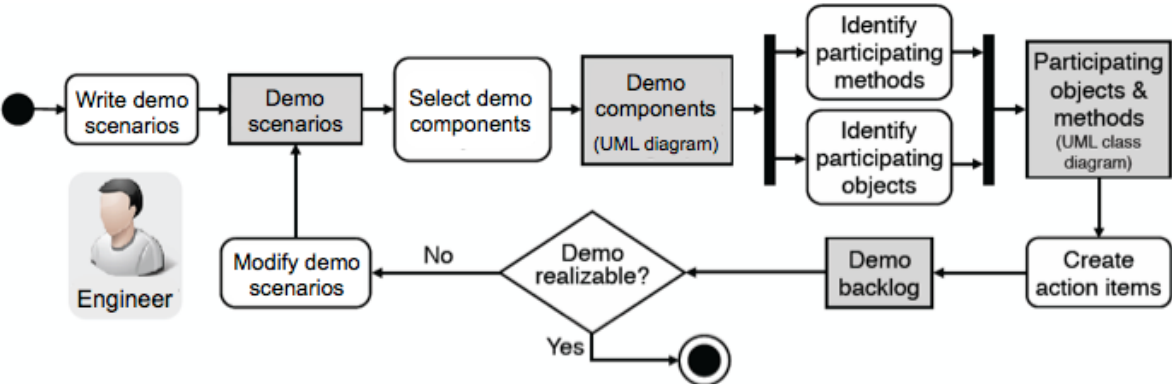


Figure 3.7 Creating demo scenarios and the demo backlog (adapted from [Krusche2018])

In order to identify participating objects and methods associated with the demo components, an effective way is to inspect the flow of events in the relevant demo

scenarios. In practice, one could use textual analysis methods, e.g., Abbott's technique [Abbott1983], to "identify nouns as candidates for classes (participating objects) and verbs as candidates for operations (participating methods)" [Krusche2018].

As mentioned previously, the goal of Demo Engineering is to create demo systems for the purpose of demonstration in a faithful and reliable way while keeping the development cost minimal. A major difference between demo-oriented development and other development methods is that not every object and method needs to be faithfully implemented. Real implementations are only for demo components that have introduced new technologies; thus, the feasibility of the new technologies could be reliably evaluated. For other components, such as those that try out only new concepts and new user experience [7], mock implementation is sufficient for the evaluation of their applicability and usability. For components that have not introduced any innovative tryout, they are normally not considered to be part of the demo system, unless they are necessary for supporting the demonstration as participating components.

In order to create the mockups, we could use *mock object pattern* [Mackinnon2000], where real collaborators (i.e., participating objects) are replaced with mock collaborators (see Figure 3.8) [Krusche2018]. It enables the project team to focus only on the real implementations of identified objects and methods and to mock other parts of the system that are not relevant [Krusche2018]. The *dependency injection pattern* [Martin1996] enables the switch between mock and real implementations during the development [Krusche2018]. It is suggested that mocked subsystems, objects, and methods in the class diagram should be highlighted so that all the team members could be aware of the decision [Krusche2018].

[7] Certain design of user experience may rely on specific enabling technology to achieve the desired effect. In such cases, since the usability of the user experience is decided by the feasibility of a particular technology, the demo system should include the associated demo component for real implementation as well.

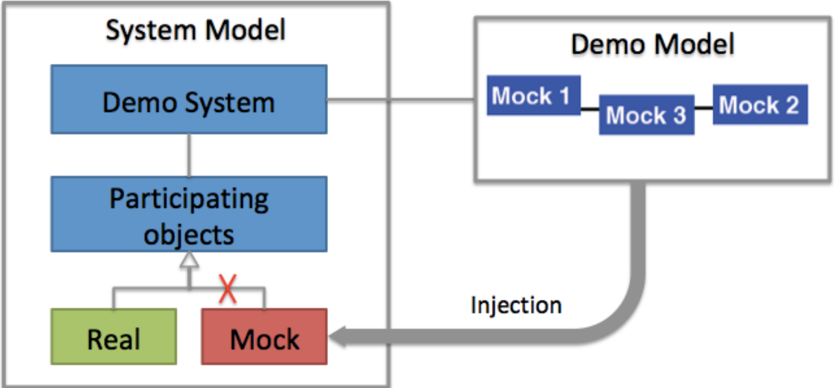


Figure 3.8 Model-based demonstration using the mock object pattern (adapted from [Krusche2018])

The implementation of the demo system could follow any continuous integration and continuous delivery process, such as Rugby [Krusche2016]. It should be noted that the implementation process, like any other software development, is never supposed to be a linear process. The demo backlog and demo scenarios are likely to be modified as the project goes on, especially when it passes milestone points (e.g., when the implementation of an important component is finished).

3.3.3 Software Theater Presentation

The presentation is performed using the demo system according to the *theater script*. The **theater script** (or screenplay) is created based on a demo scenario, which reflects the same flow of events but is written from a theatrical perspective. Performing the demonstration in Software Theater is not just going through all the functions of the demo system; the mood and atmosphere created by the actors, stage setup, and props are also essential in delivering a successful demonstration. Apart from the flow of event, details about how the actors should behave (e.g., giving the right mood), how the stage should be prepared (e.g., intended atmosphere and scene) and what props are required (e.g., to enhance the performance effect), should be considered and recorded in parallel to writing the flow of event.

Table 3.1 A typical plan for review meetings (from [Bruegge2010, p113])

Review	Date	Deliverable (release due 1 week before review)
Client review	week 7	Requirements Analysis Document
System design review	week 9	System Design Document
Object design review	week 13 (2 sessions)	Object Design Document
Internal review	week 16	Unit and integration tests
Client acceptance test dry run	week 17	All project deliverables
Client acceptance test	week 17	All project deliverables

A good opportunity to perform Software Theater is at review meetings because review meetings are organized according to the predefined milestones, and all the stakeholders (including the customers) are supposed to be present. According to Bruegge [Bruegge2010, p113], a typical plan for review meetings with stakeholders is arranged like Table 3.1. For exploratory projects using Demo Engineering, a common arrangement is to perform Software Theater at two review meetings: *design review* and *customer acceptance test*. At the *kick-off meeting*, since it is the starting point of the project, the main focus is requirement clarification and problem statement, and the form is mainly text, pictures, and videos.

- At the *design review*, since the project team have settled the requirements and worked out the main design after having explored different possibilities [8], Software Theater could be used for illustrating and evaluating the applicability of concepts and the usability of user experience. For certain enabling technologies, their technical feasibilities could also be evaluated at this phase, if the relevant demo components have been implemented. For participating objects and methods

[8] Given the fact that "change is the only constant" as well as the nature of exploratory projects, settled requirements and worked-out design have the possibility of being reverted and reworked again.

without innovative tryouts, mock implementations are enough. Videos could also be used as an alternative to live demonstrations at design review (see Section 3.2.1).

- At the *customer acceptance test*, as the demo system has already been implemented, Software Theater performed at this phase focus on the integration testing of the applicability of concepts, the usability of user experience, and the feasibility of enabling technologies. It is important that demo components with innovative tryouts be faithfully implemented to ensure a reliable evaluation. For example, the *response time* is a key performance indicator for object recognition features. Therefore, this part of the demonstration should be highlighted during the demonstration. To this end, a scenario about object scanning and recognition should be included in the demo scenario and the theater script should have a footnote such as "when the actor starts to scan an object using the App, the *second screen* [9] should give a close-up view of the App so that the system response could be clearly observed by the audience".

Table 3.2 Presentations at different phases of a project

Milestone	Purpose	Content
Kick-off	Requirements clarification and problem statement	Text, Pictures, Videos
Design review	Illustration of concepts, user experience, and some technologies	Mockups, Implementation
Customer acceptance test	Integration testing of concepts, user experience and technologies	Mockups, Implementation

[9] It is suggested to set up two screens on the stage, showing different worlds (subject world, system world, and usage world) in the parallel. See Section 4.2.1 for more details.

After the demonstration, the project team collects feedback from the audience, which will be used as input for the next iteration. The artifacts generated from the workflow, such as visionary scenarios, formalized scenarios (including demo scenarios), and theater scripts, could be modified according to the feedback, which will lead to the next version of demo system and the next round of demonstration. Figure 3.9 shows all the artifacts generated from the workflow in an analysis object model and illustrates how these artifacts are related to each other. For instance, the action items are connected to participating methods that need to be implemented for the demonstration.

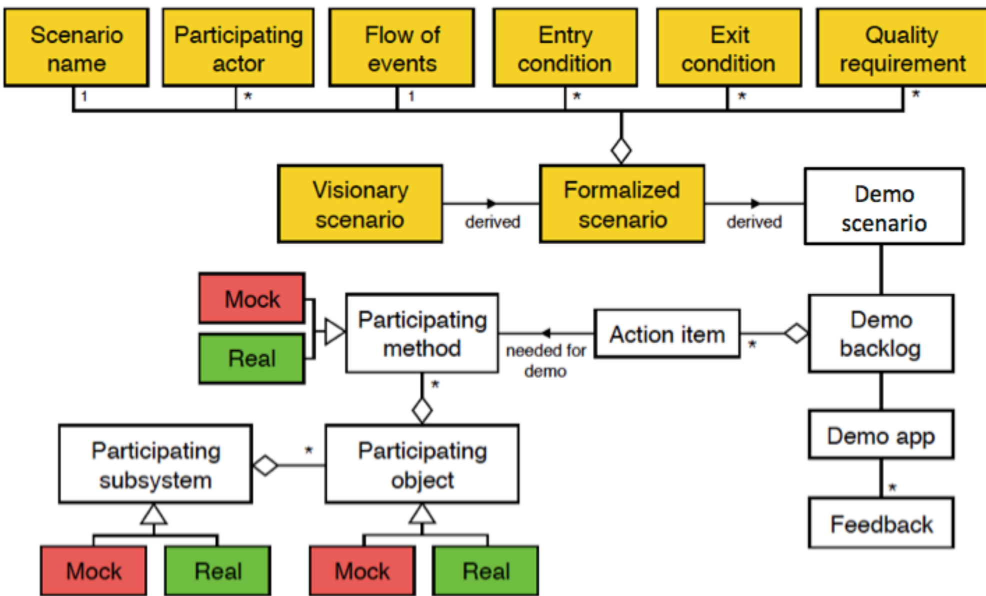


Figure 3.9 Artifacts generated from the workflow of Demo Engineering (adapted from [Krusche2018])

It should be noted that the workflow described above represents only one work cycle of the project. In real-world projects, it requires multiple cycles to close an exploratory project, depending on the size of the project.

3.4 Principles of Demo Engineering and Software Theater

This section describes the principles and supporting theories behind Demo Engineering and Software Theater.

3.4.1 Co-development of the user model and the system model [10]

In the following, we review the *user model* and the *system model* (refer to Section 2.1.3) according to the *yin-yang principle* and, on top of that, point out that user model and system model depend on and transform to each other, finally conclude that user model and system model should be given equal attention and be co-developed in user-centered software development.

Yin and yang are concepts that originated from the ancient Chinese text *The Book of Change* [11] [Cleary1992]. The Book of Change, as its name suggests, describes the philosophy of change in terms of yin and yang. Yin and yang could be understood at different levels: *two modes*, *four forms*, and *eight trigrams* [Cleary1992].

- *Two modes* describe the static properties of yin and yang, as the two sides of duality: yin and yang are ubiquitous (e.g., birth and death, heaven and earth, good and bad) [12]; yin symbolizes feminine, dark and passive, while yang symbolizes masculine, bright, and active; yin and yang are opposite but complementary [13]. [Chan1963, Cleary1992, Fang2012]

[10] This section is abridged and adapted from the Section 2 of [Xu2013].

[11] Also known as *I Ching* or *Yi-jing*. They are different translations referring to the same thing.

[12] It should be noted that different pairs of yin and yang could be identified for the same thing when looking at it from different perspectives.

[13] It should be noted that being opposite just means that they are two sides of the duality and should not be viewed as logically contrary; they are complementary to each other instead.

- *Four forms* reveal the dynamic nature of yin and yang: yin and yang depend on each other and transform to each other in a continuous manner; every yin comes with a bit of yang, and every yang comes with a bit of yin. Therefore, we have four forms of yin and yang: *climaxing yin*, *climaxing yang*, *incipient yin,* and *incipient yang* [Cleary1992].

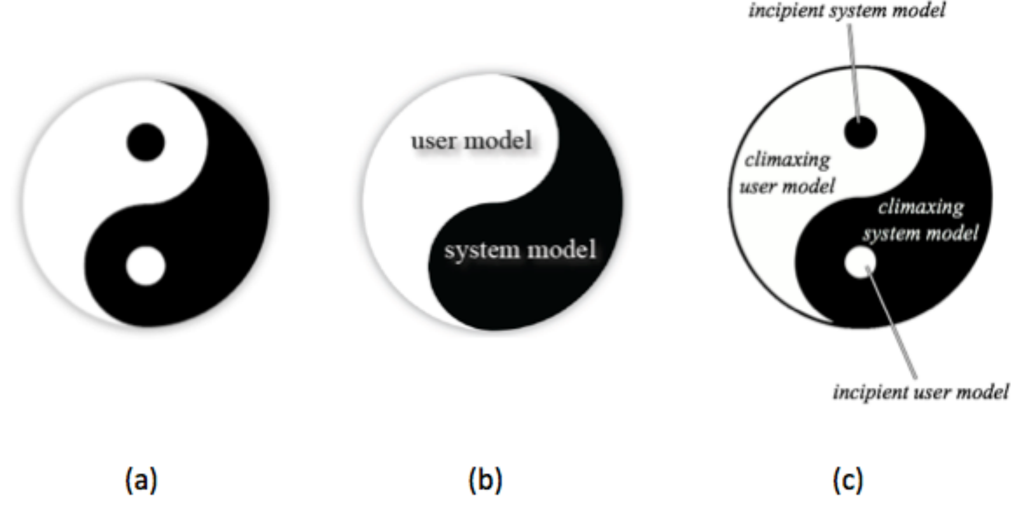


Figure 3.10 Yin and Yang shown in a Taiji diagram (from [Xu2013])

The *Taiji* diagram (see Figure 3.10a) depicts the *two modes* and *four forms* of yin and yang. The circle is divided into two halves by an S-line: the white area represents the *yin mode*, and the black area represents the *yang mode*. The yin mode comprises climaxing yin and incipient yang (the black hole in the white half), and the yang mode comprises climaxing yang and incipient yin (the white hole in the black half) [Cleary1992]. The philosophy of yin and yang is all about harmony and balance during the development of things, which could also be taken as reference for the yin and yang in software development. In our opinion, the user model and system model are the yin and yang in user-centered software development:

- User model, reflecting the user's expectation and understanding of the system, is the yin; and system model, representing the underlying implementation of the system, is the yang (as depicted in Figure 3.10b).

- User model and system model are opposite because one is concerned with external, user-visible aspects of the system, and the other is concerned with the internal, user-invisible aspects of the system.
- User model and system model are complementary to each other because the internal mechanics and external experience together constitute a complete model of the software system.
- User model and system model depend on and transform into each other constantly. System model depends on user model in that the system model is created based on the requirements reflected in the user model. From another perspective, user model transforms to system model because, in the software engineering process, user requirements reflecting the user model will gradually be transformed into the implementation that reflects the system model. In order to explain how system model transforms into user model, it is necessary to mention that certain changes, despite being unavoidable, could be made to happen earlier by speeding up the iteration cycle (see Section 3.1.1). For example, users need to see the working system to build their user models, and then to give feedback that might lead to further improvement of the system. It is common that, after having tried out an early version of the system (*incipient yang*), the users would change their minds and propose further requirements (*climaxing yin*) [Cohn2004]. Therefore, the end of one iteration would be the start of the next iteration. In other words, system model facilitates the refinement of user model, and user model again pushes further improvement of system model. This explains the process of system model "transforming" into user model (see Figure 3.10c).

The duality between the user model and system model reminds us that the user model and system model should be co-developed and receive equal attention during user-centered development. This insight also reflects the following principles of Demo Engineering:

- Demo Engineering highlights the importance of user involvement. Presenting demo systems using Software Theater enables users to "touch and feel" the systems in an engaging, sympathetic, and tangible manner, which is helpful in generating high-quality feedback for further refinement of the systems.
- Demo Engineering emphasizes the faithfulness of demo systems. Only when new ideas being demonstrated with sufficient implementation, would it be possible to make evaluation of these new ideas reliably.
- Demo Engineering recognizes the interplay between different levels of the system and advocates the integration testing of new concepts, new user experience, and new technologies in an theatrical way.

3.4.2 Using prototypes with scenarios

As mentioned in Section 2.2, some authors considered scenarios and prototypes as two "points" on the same spectrum and would represent scenarios (in a broad sense) with different **Mediums** [Nielsen1993, Rolland1998]. In this dissertation, we hold the opinion that scenarios and prototypes are relevant but different concepts (focusing on different aspects of the system) and should be used together.

Generally speaking, **scenarios are stories about use**, which describe a procedure or a process of how to use the system to achieve specific goals. **Prototypes, however, are simulations of the full-fledged future system** [Bruegge2010, p.43]. As Nielsen described, "the entire idea behind prototyping is to save on the time and cost to develop something that can be tested with real users." [Nielsen1993, p.94] A prototype is intended for being used for testing purposes but does not describe the usage on its own. Anton and Potts pointed out a significant difference between scenarios and prototypes: an executable prototype is a *behavior generator*, and a scenario is a *behavior description* [Anton1998]. That is, a prototype is an approximate representation of the future system with limited function and fidelity. A prototype requires a certain amount of implementation (or

simulation), at least for the part under evaluation, so that the user could experiment with that part and give feedback. Prototypes enable technical evaluation of the system to a certain extent, depending on the fidelity of the prototype. On the other hand, a scenario is a description about the usage of the system (which could be simulated by prototypes) in the context from the user's perspective. Evaluation with scenarios focuses on the concept and user experience that are tangible to users.

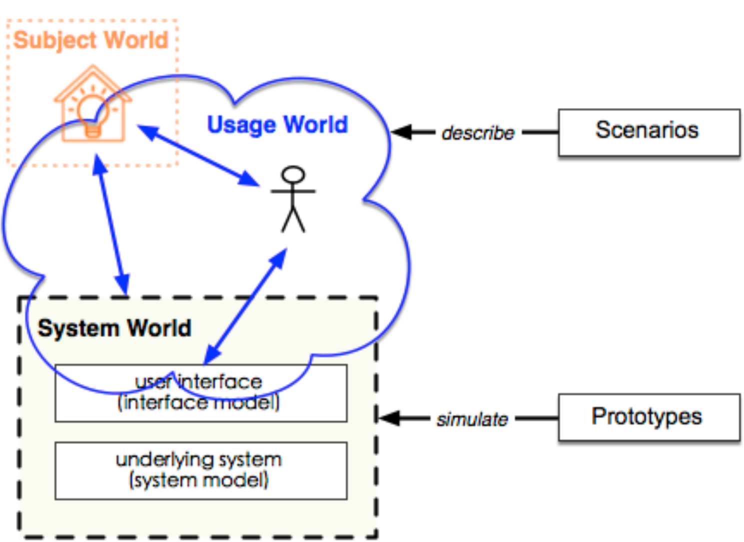


Figure 3.11 Prototypes and scenarios depicted in three-worlds conceptualization

As depicted in Figure 3.11, the relationship between prototypes and scenarios could be clarified in terms of the three-world conceptualization (refer to Section 2.1.2): prototypes simulating the future system are about the *system world* and scenarios describing the interaction between the user and the system in the context are about the *usage world* [Ben1999].

The purpose of creating prototypes is for users to experience and evaluate how the future systems would look like, so that users could provide valuable feedback for the next iteration. In practice, prototypes are often used together with scenarios [Weidenhaupt1998]. As Pohl describes, "when presenting the prototype to the

stakeholders, the usage scenarios are provided as tasks which the stakeholders shall perform with the prototype" [Pohl2010, p.459]. Compared to communicating with stakeholders with text-based scenarios in a "dry" way, illustrating future systems using prototype-based scenarios is more intuitive and easy to understand. There are different ways to use prototypes with scenarios. A prototype can be used with a guide or without a guide. It could be either *user-performed* [Pohl2010, p. 459], or *developer-performed* [Sutcliffe1997]. As Weidenhaupt et al. reported, combining the development of scenarios and prototypes enables stakeholders to check, discuss and update scenarios and prototypes at the ground level, and can lead to better customer satisfaction [Weidenhaupt1998]. This is particularly true when we are developing innovative applications based on emerging technologies (such as wearable computing, internet of things, and AR/VR). Demo Engineering uses Software Theater for the demonstration of future systems, which creates more empathy and context. Besides, unlike vertical prototyping or horizontal prototyping, Demo Engineering using Tornado Model focuses on faithful integration testing of concept, user experience, and technology with minimal effort, which will be described in the next section.

3.4.3 Focus and minimal effort with demo-oriented development

To address the challenges posed by exploratory projects (as described in Section 3.1), we need a demo system that "just fits" for the integration testing of new concepts, new user experience, and new technologies. In order to achieve faithful and minimal effort, the demo system should be implemented in a "no more, no less" fashion. Tornado Model is a demo-oriented development method that aims to develop such demo systems continuously in different phases of the development project [Bruegge2012, Bruegge2015]. The basic tenet is to start with low-fidelity prototypes for potential tryouts on the long list and employ high-fidelity prototypes for those selected on the shortlist. This is also following the spirit of *idea funnel* (see Figure 3.12), which has been used for effective idea management in innovation management [Luecke2003, Pikkarainen2011]. In an *idea funnel*, ideas are evaluated and screened using different techniques at different phases

[Luecke2003, p.62; Dabholkar2013, p.15]. At the top of the funnel, since there is a heap of potential ideas under consideration, low criteria is applied to narrow the list of candidate ideas quickly. As the ideas go through the funnel, only a shortlist of ideas survives at the bottom of the funnel. Since only a small amount of ideas are left now, high criteria could be adopted to make further in-depth evaluation [Kohavi2013]. This is somehow consistent with the spirit of *lean startup* as well, which advocates evaluating ideas with the *minimal viable products (MVP)*. [Ries2011]

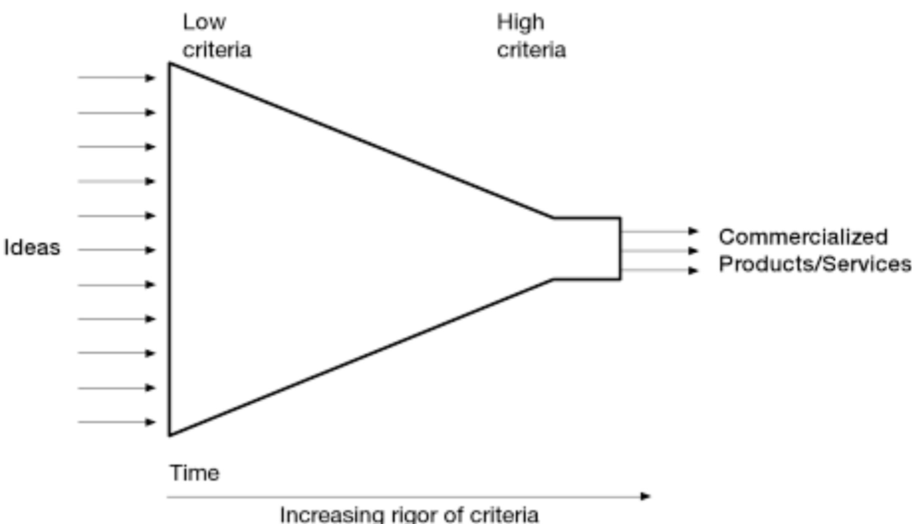


Figure 3.12: Idea funnel for effective idea management (from [Luecke2003])

The spirit of Tornado Model is also similar to that of prototyping techniques in the sense that they both emphasize the implementation of selected parts of the future system. A major difference between them lies in how to make such selections. While prototyping can be vertically, horizontally, or mixed (see Figure 3.13), Tornado Model stresses more on "focus" and "no more, no less". To be specific, Tornado Model only implements those components that are required for the demonstration: real implementations are used for where faithfulness is important, such as those components using new technologies (whose feasibility needs to be evaluated); and mock implementations are used for participating objects without innovative tryouts.

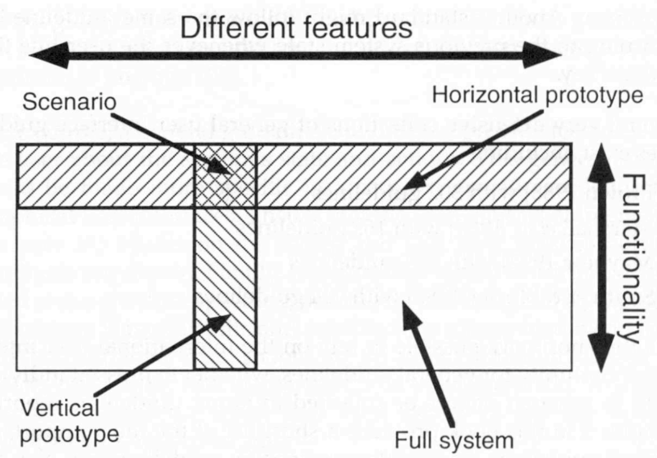


Figure 3.13 Vertical prototype and horizontal prototype (from [Nielsen1993, p.94])

3.4.4 Using theatrical techniques for the demonstration

As mentioned in Section 3.1, exploratory projects are to develop innovative systems that are not seen in the market (for instance, using new interaction paradigms like gesture in the air, eye control, etc.). For situations like that, text-based scenarios and prototypes are not enough because they cannot highlight the innovative tryouts of the future system in a lifelike context and therefore cannot deliver the empathy and "touch and feel" to the users, let alone makes a reliable evaluation on the feasibility of the new technologies. In other words, for these exploratory projects, new concepts and new user experience lead to a situation that the user model cannot be rapidly constructed by just watching the changes of the user interface in the system world.

Using prototypes with scenarios (see Section 3.4.2) has been proved to be a useful way to enhance design and user participation in the demonstration [Weidenhaupt1998, Sutcliffe1997, Sutcliffe2013]. In a step further, theatrical techniques could be applied to the demonstration of software systems [Mahaux2010, Xu2015, Krusche2018]. Needless to say, the cost for performing demonstrations in Software Theater is higher than writing

text-based scenarios; however, it is worth considering the benefits of using theatrical techniques than text-based scenarios as well as the significance of the exploratory projects. Like text-based novels, the performances in films and theaters are all about telling stories. Actually a performance and a novel could tell the same story in terms of the *materials of the story* (which is consist of the plot, characters, setting, theme, etc.); however, they differ in the *form of narration*: novel is mainly based on written language and performance is based on the acting of the actors as well as pictures and sound [Monaco2013, p.54] [14]. According to Monaco [Monaco2013], this essential difference makes further distinctions between performances and novels: performances are more life-life and create a much rich experience for the audience; performance leaves more room for the audience to interpret the connotation, as compared to novels where "we see and hear only what [the author] wants us to see and hear", because for performances, the audience "see and hear a great deal more than a director necessarily intends" [Monaco2013, p.54].

Demonstrations can be considered as a proof-of-concept method that shows functional and non-functional aspects of the future system to the stakeholders and inspires feedback from them. Basically, it is a process of establishing synchronization between the *user model* and the *design model* (refer to Section 2.1.3). Demonstrations can be used to verify not only requirements but also other aspects such as software design, accessibility, and usability, etc. Ideally, a demonstration should be carried out whenever there is a major change that may cause uncertain consequences; however, in practice, due to the cost reason, the performance of demonstration only happens at milestones. In our opinion, no matter a demonstration is based on a partially-implemented prototype or a fully-implemented system, putting the demonstration into a *usage world* is helpful, especially for applications with novel interaction paradigms such as wearable computing applications and IoT-based smart home systems. As mentioned in Section 3.4.2,

[14] The original description in [Monaco2000] was about films. Although there are differences between films and theatrical plays, the citations used in this section do not go beyond their commonalities.

scenarios stimulate more and deeper discussions among stakeholders [Sutcliffe1997, Pohl2010]; presenting scenarios in Software Theater strengthens this benefit because it is more intuitive and tangible to the audience. When the actors present the demo system according to the theater script in a lifelike scene, people in the audience are virtually placed "on the scene" and could "touch and feel" the future system, which will generate more empathy and lead to more insightful feedback. Besides, Software Theater employs theatrical techniques and film tricks during the performance, which makes it easier to highlight the existing problems (e.g., the pain points of the user) and the benefits of the future system. Furthermore, with the aid of stage equipment (such as projectors and audio effects), it is possible to perform stage montage to achieve expected effects that are not viable in other forms of demonstration due to technical barriers.

3.5 Related Works on Software Theater

This section describes related works that also employ certain kinds of live performances. We will also compare them with Software Theater and point out the differences.

3.5.1 Laurel's Computers as Theatre

In the book *Computers as Theatre* [Laurel2014], Laurel elaborated on the dramatic foundations and highlighted the linkage between theater and user interaction design. According to Laurel, "we have at least two reasons to consider theatre as a promising foundation for thinking about the designing human-computer experiences. First, there is significant overlap in the fundamental objective of the two domains - that is, representing action with multiple agents. Second, theatre suggests the basis for a model of human-computer activity that is familiar, comprehensible, and evocative." [Lauren2014, p.30] However, Laure's *Computers as Theatre* only focused on using the heuristics learned from theater to form general rules for user interaction design and did not enact the theatrical performance itself; nevertheless, its description of the dramatic

foundation, e.g. the story structure [Laurel2014, p.96] and the dramatic devices [Laurel2014, p.105] is enlightening to this dissertation (see Section 4.1).

3.5.2 Mahaux and Maiden's Improvisional Theater

Mahaux and Maiden proposed using *Improvisional Theater* to support team-based innovation in the requirements engineering process [Mahaux2008, Mahaux2010]. The commonality of Improvisional Theater and Software Theater is that they both employ the form of theater in order to improve communication and increase mutual understandings with stakeholders. But they differ in several aspects. First, the purpose of Improvisional Theater is to generate creative ideas in the requirements engineering process, while the purpose of Software Theater is to demonstrate and evaluate future systems developed under exploratory projects, including the applicability of new concepts, usability of new user experience design and feasibility of new technologies. Second, Improvisional Theater, as its name suggests, takes advantage of *unplanned* improvisional performance to stimulate the creativity of team members, while Software Theater uses a predefined theater script for the demonstration.

3.5.3 Role-playing based on CRC Cards

The *CRC (Class, Responsibilities, and Collaborators) Cards* were introduced by XP practitioners Kent Beck and Ward Cunningham in 1989 as a teaching tool for object-oriented programming. [Beck1989] Some also employ CRC cards for role-playing where students are asked to play the role of specific classes and interact with other each, as an effort to understand the responsibility of a class and how a class interacts with another. While *CRC cards* method shares some commonality with Software Theater in that role-playing is similar to acting a role in a theatrical play; however, they are different in multiple aspects. In terms of purpose, role-playing using CRC cards is designed for teaching object-oriented programming, while Software Theater is intended for the demonstration and evaluation of future systems. Besides, CRC cards represent classes,

and the role-playing activities illustrate the behavior of the classes, while in Software Theater, the actors illustrate the future system by interacting with the demo system.

3.5.4 Rice and et al.'s Forum Theatre

Rice and et al. used *forum theatre* (a kind of interactive theatre) to elicit requirements in the development of new technologies [Rice2007]. They also discovered that *storytelling using theatre and video* is useful in promoting user involvement because it is easier for technologically naive users (for instance, the elders) to understand future systems, and it "can increase designer empathy towards end users". [Rice2007] However, as Rice and et al. pointed out that "we found theatre particularly useful at the stage of the requirements gathering, it may not be equally well suited for very early requirements gathering, or later, more specific prototype evaluation" [Rice2007]. The main reason for this discrepancy is that Forum Theatre focuses only on requirements elicitation, while Software Theater is intended for the demonstration and evaluation of a functional demo system. Besides, it is related to the targeted types of projects. Software Theater is used for exploratory projects with both novelty and uncertainty. The live demonstration is partially to address to challenges brought by the uncertainties, and the cost is worth compared to the novelty achieved in the end.

Chapter 4
Software Theater Patterns and Best Practices

"I have not failed. I've just found 10000 ways that won't work."

-- *Thomas A. Edison*

We have elaborated on the concept and principles of Software Theater in Section 3. This chapter first introduces the heuristics that we have learned from the film and theater theory, then on top of that describe the rules that we should follow when using Software Theater. Next, based on these rules, we describe Software Theater patterns and best practices, where *patterns* refer to the typical recurring design of performance for the specific context (similar to the *tropes* used in novels and films) and *best practices* refer to tips and hints that help make appropriate use of Software Theater in other respects.

4.1 Heuristics from Theater Theory

Software Theater demonstrates software systems by live performance on the stage. Looking into the theories behind theater helps us to take full advantage of its strong points and to overcome its limitations.

4.1.1 Forms of the performance

Theater and film are similar forms of performance in the sense that they both are performed by actors according to the screenplay, although films sometimes are based on narration and pictures (such as documentation films). As Cooper and Dancyger described in *Writing the Short Film*, "as a narrative form, [film] has more in common

with theater." [Cooper2005, p.101]. However, apart from this common point, the characteristics of theater and film differ in several aspects [Monaco2013]:

- Theatrical plays are more live and interactive than films, bringing more immediacy and intimacy [Monaco2013, p.60]. As Monaco described, "theater has one advantage over film, and it is a great one: theater is live. [...] It is also true that the people who perform in film are, quite obviously, not in communication with their audience". [Monaco2013, p.58]
- Films could inherently support more forms of mediums and show more details than theatrical plays. [Monaco2013, p.57] For example, a film could use techniques like close-up and sound effect, etc. to enhance the visual and aural experience of story telling. But for a theatrical play on the stage, the audience "has difficulty comprehending all but the broadest gestures." [Monaco2013, p.58]
- Compared to theatrical plays that are based on performing a series of events, films could carry more communications to the audience. [Monaco2013, p.58] For example, the director of a film could insert voice-overs to explain the story or use close-up lens to highlight crucial details of the film.

Software Theater is performed based on theater but does not need to follow every conventional restriction on theatrical plays. Recognizing these differences helps us to make appropriate use of theater for the purpose of demonstrating software systems.

- In this dissertation, we have advocated using theatrical plays for demonstrating software systems. However, there are cases where a Software Theater cannot be performed due to technical, spatial or geographical reasons. In such cases, film-based *Software Cinema* [Creighton2006] could be used as an alternative. As mentioned above, films are less interactive than theaters, but the expressive power is beyond theaters. (See Section 4.3.3 for the best practice of *using video as an alterative to theater*)

- Due to the limitation of the stage, conventional theaters suffer from the missing of visual details in the audience because the fine details on the stage cannot be clearly observed by the audience. In Software Theater, it is crucial for the audience to see the changes on the user interface while the actors are demonstrating the system on the stage. To enhance the audience's visual perception, it is suggested to set up at least one projection screen above the stage. This screen could be used either for close-up of details on the stage or sharing the screen of mobile phones or computers to enable the audience to know the status of the *system world*. (See Section 4.3.1 for the best practice of *Setting up additional screens for synchronized illustration*)

- Theatrical plays normally do not use voice-overs (because it would interrupt the storytelling on the stage) and therefore carry less communication to the audience. However, using voice-over is not a problem when we are demonstrating software systems; besides, for short performances, narration is more efficient in introducing the background and developing the story because "there is too little time available [...] to allow the action to develop on its own." [Cooper2005, p.146] (See *Narrator* pattern described in Section 4.2.2)

4.1.2 Story and storytelling of the performance

The same materials of the story could be performed in different manners and result in different qualities of the performance. The story is one thing and the storytelling is another - both are important for delivering distinctive and impressive performances. First, we look at the storytelling. In the theory of films and theaters, there are different well-known structures for storytelling, including **Freytag's 5-point structure** (*Introduction*, *Rise*, *Climax*, *Return or Fall*, *Resolution*) [Freytag1898; Laurel2014, p.96], **Watts's 8-point arc** (*Stasis*, *Trigger*, *The quest*, *Surprise*, *Critical choice*, *Climax*, *Reversal*, *Resolution*) [Watts1996; Caldwell2017, p.14], and **Campbell's 12-point journey** (*Ordinary World, Call to Adventure, Refusal, Meeting with the Mentor, Crossing the Threshold, Tests/Allies/Enemies, Approach to Inmost cave, Ordeal, Reward, The road*

back, Resurrection, Return with elixir) [Campbell2004; Vogler2007, p.7], etc. Although above-mentioned structures, among others, have different points and definitions, they could be mapped to a common model of three acts: *Act 1*, *Act 2*, and *Act 3* (see Figure 4.1) [Caldwell2017, p.18]. Caldwell summarized what these acts do:

- "**Act 1** - the setup introduces the characters and the rules of the world. The audience/player learns where the story takes place (the setting), what the main character wants (motivation), and the dramatic question (what the story is really about, that the audience can relate to). This act contains only the minimum amount of information the audience needs to start the story.
- **Act 2** - Increasing Conflict forces the main character to confront obstacles that stand between them and what they want, their goal. These conflicts build until the final crisis that has to be resolved… one way or another. This act is where the bulk of the conflict takes place.
- **Act 3** - Resolution follows the Climax that is the transition to Act 3. Here, the conflict is resolved, the big questions are answered, and a new status quo is established. It's the shortest act, with a resolution which gives the story its meaning." [Caldwell2017, p.8]

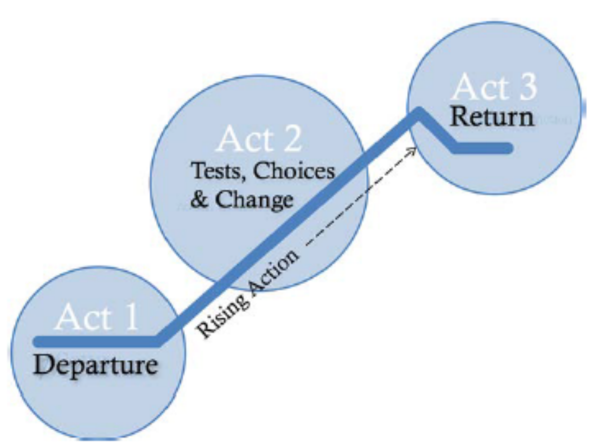


Figure 4.1 The three-act structure (from [Caldwell2017, p.8]

In our opinion, the performance used for Software Theater is short compared to a regular film or theater, and could take this three-act structure as a reference framework:

- **Software Theater Act 1** introduces the background about the project (e.g. why the project? what is the purpose of the system? who is supposed to use it? where the system is used, a hospital, or a factory?) as well as the actors (e.g. their roles, goals, and personalities). Although the context information is not the main part of the story, it foreshadows the conflict in the next act. As Cooper and Dancyger described, "character, plot and conflict are intricately related to one another. [...] Plot qualities are closely related to character. [...] The more powerful the barriers that stands in the way of the character achieving his or her goal, the more compelling the plot. If the character faces no barriers in achieving his or her goal, there is no story." [Cooper2005, p.91] Giving a clear explanation about the actor's task and goal helps the audience to understand his or her difficulty with existing system and makes it natural to anticipate the new system. To this end, we could employ voice-over or screen illustration to provide additional information to the audience. (See Section 4.2.2 for the *Narrator* pattern and Section 4.3.1 for the best practice of *Setting up additional screens for synchronized illustration*)

- **Software Theater Act 2** illustrates the interaction with the system for achieving specific goals. In order to create dramatic effects, storytelling devices, e.g. conflict and twist, could be applied to maintain the attention of the audience. Conflict and twist are general concepts that can be embodied in different concrete forms. According to Cooper and Dancyger, *conflicts* could be created from the actor's perspective: actor vs. actor (e.g. different opinions and demands between actors), *actor vs. setting* (e.g. the external difficulty for the actor to achieve the goal), *actor vs. community* (e.g. the actor advocating the new system as opposed to others defending for the existing system) [Cooper2005, p.120]. In Software Theater, *contrast* between the existing system and the new system could be considered as yet another form of conflict. *Twists* could be embodied as *turns*, *surprises*, *reversals*

[Cooper2005, p.106; Laurel2014, p.105] as well as *shock* [Caldwell2007, p.91] and *coincidence* [Caldwell2017, p.200], to name just a few.

- **Software Theater Act 3** concludes the performance with the conflict and the twist resolved. Since Software Theater is to demonstrate a new system, it is suggested to create an atomsphere of happy ending: the task is successfully completed and the actors are satisfied with the result.

Next, we look at the story. In the context of films (especially feature films) and theaters, screenwriters start with basic ideas of the story, then attach dramatic elements to the story to make it more interesting and attractive, and finally adapt it into the form of screenplay [Cooper2005, p.113]. However, in the context of Software Theater, we start with *demo scenarios,* which describe a sequence of event illustrating the usage of the demo system. Compared to stories in films and theaters, these demo scenarios are more about factual information and therefore require more effort to be made dramatic. On the other hand, considering the purpose of Software Theater, these "usage scenes" derived from demo scenarios are indispensable to the performance. Therefore, we distinguish between two dimensions concerning with the performance of Software Theater, which are labelled as the *core of the story* and the *drama of the story*.

- **Core of the story** is the tenet of the story that the performance must convey to the audience. In the context of Software Theater, the core of the story is a sequence of events illustrating the usage of the demo system. These "usage scenes" are crucial in fulfilling the goal of Software Theater - demonstrating the applicability of the concept, the usability of the user interaction and the feasbility of the technology. These "usage scenes" by themselves constitute less dramatic elements and are more about factual information. There are several patterns and best practices for passing factual information to the audience in Software Theater: voice-over, monologue, dialogue, metaphor, and screen illustration, etc. (see Section 4.2 for *Dialogue* pattern, *Narrator* patter, and *Metaphor* pattern; and Section 4.3.1 for *Setting up additional screens for synchronized illustration*).

- **Drama of the story**. According to Caldwell, "dramatic stories are more than just what is happening ... they are about why things are happening and how it affects the viewer." [Caldwell2017, p.4]. Attaching dramatic elements to the story's core makes the story logically complete and attractive. As Caldwell stated, "dramatic stories are still a sequence of events, but the fundamental different is that they are a sequence of *connected* events" [Caldwell2017, p.4]. The drama of the story represents the dramatic quality of the performance, which evoke the audience's engagement and sympathy. However, it should be noted that the drama should be developed to support conveying the core of the story. In Software Theater, it means that the project team should figure out a drama that fits for the "usage scene" and lends itself to highlighting the usefulness of the system. This requires the project team to identify the features of the new system and to design the drama around the killer application scenarios of these features. To enhance the dramatic quality of the performance, dramatic devices, such as *conflict*, *twist*, *contrast*, and so on, could be employed. (see Section 4.2.5 for *Conflict* pattern, Section 4.2.6 for *Twist* pattern, and Section 4.2.7 for *Contrast* pattern)

4.2 Software Theater Patterns

Patterns have been used in software engineering as a way to outline the essential part of a solution as well as the corresponding context that the solution fits for [Alexander1977]. Following the same spirit, this section describes *Software Theater patterns* that we have identified based on the heuristics from the theater theory as well as our practical experience. These patterns could be classified into two categories: patterns for passing information to the audience, and patterns for developing dramatic stories. In software engineering, there are different forms to outline patterns, such as Alexander's schema [Alexander1977], Gang-of-four's schema [Gamma1996], and Gang-of-five's schema [Buschmann1996]. In the following, we adopt an abridged version of Gang-of-five's schema [Buschmann1996, p.20] for describing patterns in Software Theater:

- *Name*: the name of the pattern.
- *Context*: the context where the pattern emerges. It could be either *passing factual information to the audience* or *drama development for the story*.
- *Problem*: the situation and problem that the pattern addresses.
- *Solution*: the essential part of the pattern that addresses the problem.
- *Consequences:* The benefits and drawbacks of the pattern, and/or any point that needs to be noted when using the pattern.
- *Example*: an example describing how to use the pattern.
- *Source*: instead of providing a snippet of sample code, we provide a link to the recorded video of the demonstration that has used the pattern.
- *Related Patterns:* patterns that are related and could be potentially used as an alternative to the pattern in discussion

4.2.1 Dialogue pattern

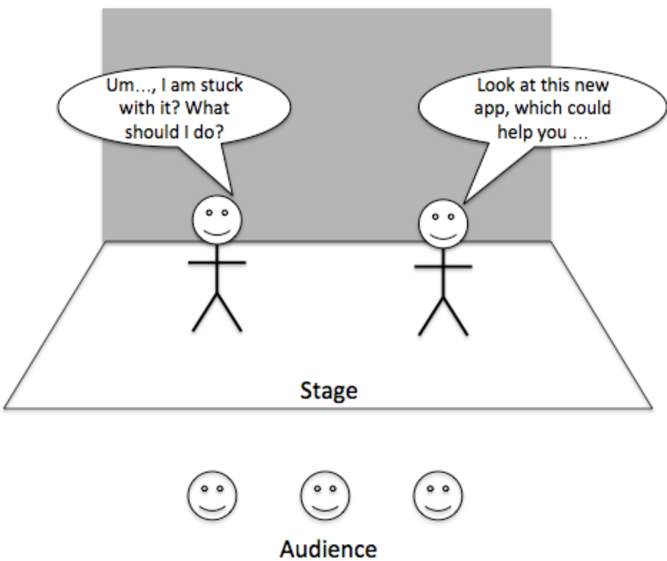


Figure 4.2 The Dialogue pattern illustrated

- **CONTEXT:** Passing information to the audience
- **PROBLEM:** Additional information is required as prerequisite knowledge for the audience to understand the demonstration, especially when the system has a new concept or new interaction paradigm that the audience has never seen before. Such information could be the background about why the system is required, what constraints the system is faced, etc.
- **SOLUTION:** The Dialogue pattern passes information to the audience by the conversation between two actors on the stage. A typical way to apply Dialogue pattern is to set up two actors on the stage: one acts as a novice, and the other acts as an expert; the novice actor deliberately asks questions about the system, and the expert actor answers these questions. The audience receives additional information through listening to the dialogue between them.
- **CONSEQUENCES:** To apply this pattern, it is important to insert the necessary information into the conversation, while writing the theater script, such that the conversation goes in a natural way fitting the whole story. This could be easily achieved in most cases, which makes the Dialogue pattern the commonly used among all the patterns in the context of information passing.
- **EXAMPLE:** The BSB Reservation Project from iPraktikum 2017/2018 [15] employed *Dialogue* pattern to introduce their IoT-based seat reservation system for the library. In this project, the team developed a mobile app for seat reservation in the library. Using the app, a user could reserve a seat in the library in advance. When he arrives in the library and finds the seat, he needs to check in by entering the check-in code of the seat in the app. The check-in code is dynamically generated for every reservation and valid for only once. Therefore, in order to show the dynamically generated check-in code, a hardware gadget (the ePaper display)

[15] Homepage of iPraktikum 2017/2018: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/937

needs to be mounted on the desk. Since this kind of interaction via a gadget was not familiar to every user in their daily life, to help the audience to understand the purpose and usage of the gadget, a novice actor asked the expert actor why there is a display on the desk and how to use it.

- **SOURCE:** https://youtu.be/VVAsaXuJPsQ
- **RELATED PATTERNS:** Narrator, Monologue, Metaphor

4.2.2 Narrator pattern

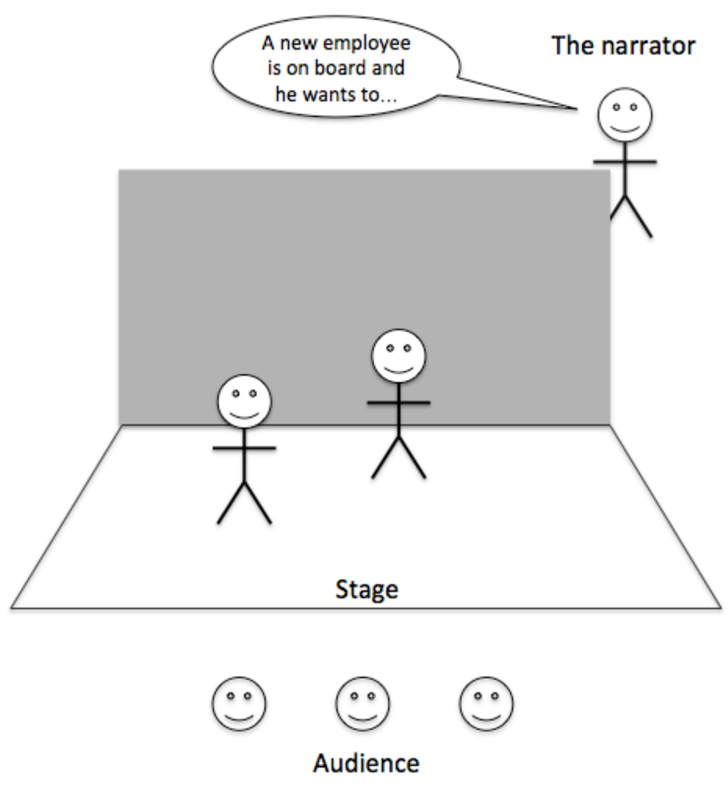


Figure 4.2 Narrator pattern illustrated

- **CONTEXT:** Passing information to the audience

- **PROBLEM:** Similar to *Dialogue* pattern, the *Narrator* pattern is another commonly used way for providing additional information to the audience. The difference lies in *who* gives voice to it. The *Narrator* pattern fits for situations where it would be confusing or unsmoothly if the information were given by one of the actors on the stage.
- **SOLUTION:** The Narrator pattern passes information to the audience by setting up a dedicated role, the narrator, who gives information to the audience in the form of voice-over and normally does not show up on the stage. For general information about the system, the narrator could explain at the beginning of the demonstration; for explanations about the usage of the system, the narrator could commentate on while the actors are interacting with the system.
- **CONSEQUENCES:** To apply the Narrator pattern, it is important to reserve an appropriate length of time in the theater script for the narration. The Narrator pattern is the mostly used pattern because the information could be easily and accurately passed to the audience without having to think about the coherence and continuity of the actor's lines. The fact that the information is pushed to the audience by the narrator, not embedded between the lines, makes it more effective in communicating to the audience. The downside is that the Narrator pattern interrupts the actors' performance on the stage and sacrifices the dramatic quality of the performance.
- **EXAMPLE:** The ZIMTlog project from iPraktikum 2016/2017 [16] developed for ZEISS Industrial Metrology (ZIMT) used the *Dialogue* pattern. The background of the project was that ZIMT, the manufacturer, would like to monitor expensive machines they produced during international transportation to its customers. In order to track the conditions of the machines, ZIMT would attach a sensor to each machine so that any shock in all three axes and deviation in temperature and

[16] Homepage of iPraktikum 2016/2017: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/949

humidity could be detected. The project was to develop a corresponding mobile app for this monitoring system so that ZEISS colleagues could use it to read the sensor data and get a first clue on where to start inspecting the damaged machines. Since the machine and the associated sensor were inside a package box, the process of retrieving sensor data was not directly visible to the audience, though it was essential for the audience to understand the whole working procedure. Therefore, a narrator was introduced and arranged to explain what was happening with the machine and sensor during intervals of the performance.

- **SOURCE:** https://youtu.be/enpEO6bGaZQ
- **RELATED PATTERNS:** Monologue, Dialogue, Metaphor, Illustration

4.2.3 Monologue pattern

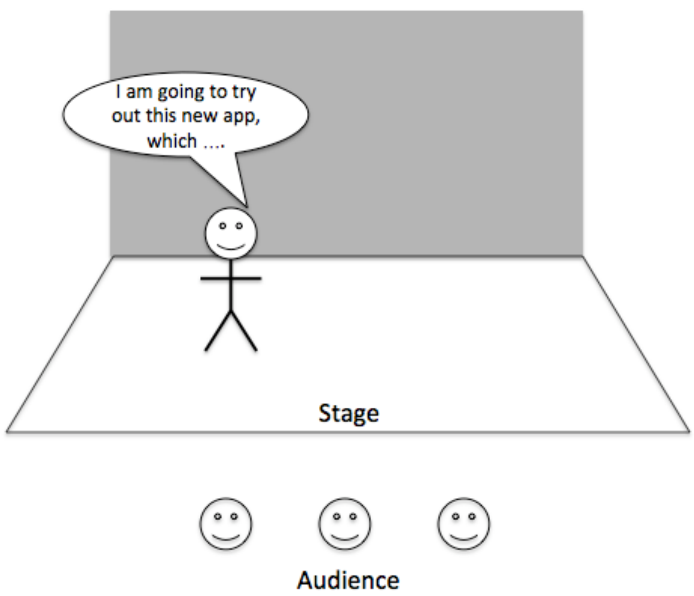


Figure 4.3 The Monologue pattern illustrated

- **CONTEXT:** Passing information to the audience

- **PROBLEM:** The Monologue pattern is yet another way of passing information to the audience. It is used when the single actor on the stage wants to communicate something to the audience, while not intending to bring another actor on the stage or to use a narrator to interrupt his or her performance.
- **SOLUTION:** The Monologue pattern passes information to the audience by talking to him or herself.
- **CONSEQUENCES:** The Monologue pattern relies on the single actor to tell everything to the audience. It should be noted that the information should be not overloaded when using this pattern. Besides, mixing the story with the background or side information could potentially confuse the audience.
- **EXAMPLE:** The NeuPro project from iPraktikum 2019 [17] developed a system to support workers to clean and repair boats by controlling a robotic arm using an iOS app. Since the Software Theater focused on demonstrating the feasibility of the new approach (using an iOS app to control the robotic arm), the scenario is plain and straightforward; therefore, the actor simply walked through all the functions on the mobile phone and explained what was happening with the robotic arm as she demonstrated the app.
- **SOURCE:** https://youtu.be/05ObdDu6FHU
- **RELATED PATTERNS:** Dialogue, Narrator, Metaphor

4.2.4 Metaphor pattern

- **CONTEXT:** Passing information to the audience, especially internal mechanism of the system

[17] Homepage of iPraktikum 2019: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/1037

- **PROBLEM:** The *Metaphor* pattern is another pattern for providing additional information to the audience but more on explaining the internals of the system, for example, explaining an algorithm or illustrating internal interactions between different modules of the system.

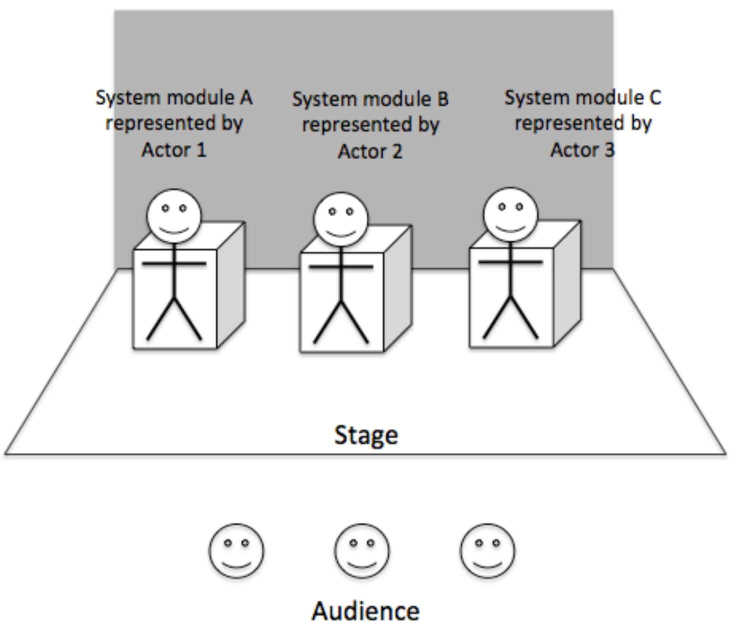


Figure 4.5 The Metaphor pattern illustrated

- **SOLUTION:** As its name suggests, the Metaphor pattern sets up fictional actors on the stage, each mimicking one module of the system, in addition to actors representing users of the system. The fictional actors could explain themselves (including their roles and their current status in the system) and illustrate how they coordinate with each other in order to accomplish a specific task.
- **CONSEQUENCES:** The Metaphor pattern could be considered as an option when you want to illustrate the internal workings of the system. The benefit is that it is easier for the audience to understand the abstract concepts behind the user interface; however, it also relies on the writing of theater script.

- **EXAMPLE:** The AllianzMan project from iPraktikum 2014/2015 [18] used the *Metaphor* pattern. The project was to develop a mobile app that would use sensors to detect illegal intrusions at home. In that demonstration, a fictional actor, "AllianzMan", was introduced as a metaphor for the home security system and further fictional actors for integral components required by the system (that is, Sensor Value Change Experts, Timeline Exports, Occupant Identification Experts, and Intrusion Domain Experts). During the demonstration, the "AllianzMan" explained to the user the main features of the system (i.e., learning the habit of the user in normal cases and triggering an alarm when an abnormal behavior was detected). Then, the "AllianzMan" and its "Experts" components coordinated somehow on the stage, mimicking status changes of the components and the interaction between components. Later, when an intrusion happened, these fictional actors cooperated with each other, ended up identifying it as an illegal intrusion, and sent an alarm to the user.
- **SOURCE:** https://youtu.be/efHEQQaVp6U
- **RELATED PATTERNS:** Narrator, Monologue, Dialogue

4.2.5 Conflict pattern

- **CONTEXT:** Drama development for the story.
- **PROBLEM:** In order to create dramatic effects, storytelling devices could be applied to maintain the attention and interest of the audience. The Conflict pattern achieves this goal by introducing conflicts between the intent and the reality.

[18] Homepage of iPraktikum 2014/2015: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/619

- **SOLUTION:** Typical conflicts in Software Theater include: 1) different actors have different opinions (the new application reconciles the conflict); 2) the actor finds it hard or impossible to achieve his or her goal (the new application helps him or her to overcome the dilemma); 3) the actor wants to do something that is assumed as impossible or inappropriate by other people (the new application convinces people to adopt the new belief).
- **CONSEQUENCES:** As a means to increase tension in the story, creating conflicts helps to attract the audiences' attention and maintain their interest. However, it should be noted that whenever you create a conflict, you would have to close the conflict by introducing a solution. The story shall not come to an end with open conflicts unresolved. Besides, considering the length of Software Theater, one or two conflicts are by experience enough for a ten-minute performance.
- **EXAMPLE:** The LocalHero project from iPraktikum 2019/2020 [19] is a community-based application that enables users to assist people who are in need of help. In the demonstration at the Customer Acceptance Test, in order to show that the LocalHero app is not only useful to people receiving helps, but also welcomed by people offering helps, an actor played the role of a retired doctor, who felt boring because he could use his expertise to help people anymore. Then, the LocalHero app was introduced to him to enable him to help people in need of medical assistance nearby. The doctor felt happy again when he helped people via receiving requests from the LocalHero app.
- **SOURCE:** https://youtu.be/B3cZeolpBLQ
- **RELATED PATTERNS:** Twist, Contrast, Straightforward

[19] Homepage of iPraktikum 2019/2020: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/1063

4.2.6 Twist pattern

- **CONTEXT:** Drama development for the story.
- **PROBLEM:** Twist is another commonly used storytelling device to create dramatic effects, especially for introducing a solution to the conflict. The Twist pattern achieves this by introducing a sudden change that is not expected by the audience.
- **SOLUTION:** Examples twists in Software Theater include *turns* (the situation gets changed by using the new app), *surprises* (the actor receives an unexpected benefit by using the new app), *reversals* (the story gets an completely unexpected ending due to the introduction of the new app), *shock* (used to intensify the problem faced), and *coincidence* (used to bridge the leap in the story).
- **EXAMPLE:** The Chargify project developed from iPraktikum 2019 [20] developed an app that helps owners of electric cars to plan the charging schedule based on the users' calendar and daily habits. To exemplify the usefulness of the app in improving the user's everyday life, the project team developed a story using surprise: the app estimated the charging time based on his travel plan in the day and turned out that the user would have enough time for breakfast; then he prepared breakfast for his wife and his wife was surprised and thanked for his arrangement.
- **CONSEQUENCES:** Twist could be positive and negative. In Software Theater, negative twists could be used for introducing problems and positive twists for putting an end to the conflict. Similar to the Conflict pattern, the use of twist is suggested to be limited to once or twice for a ten-minute performance.
- **SOURCE:** https://youtu.be/G33uFJSfaQk
- **RELATED PATTERNS:** Conflict, Contrast, Straightforward

---

[20] Homepage of iPraktikum 2019: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/1037

4.2.7 Contrast pattern

- **CONTEXT:** Drama development for the story.
- **PROBLEM:** An effective way to highlight the benefit of a new system is to compare it with the existing system, illustrating the significant differences in a everyday life context.
- **SOLUTION:** A common way to use Contrast pattern is to show how a daily routine is carried out in existing system, expose the drawbacks, and create the conflict; then, introduce the new system and show how everyday life gets improved.
- **EXAMPLE:** The Zeyes project from iPraktikum 2016/2017 [21] developed an app to support the users to do inventory management in an easier way, e.g. counting the available items using bar codes or RFID tags. To highlight the benefit of this new system, the project team set up a comparison between the existing method and the new method. In the demonstration, two actors performed the same task (counting items) in the parallel: one using manual counting and the other using the app. The result turned out that exiting method was slow and awkward, and the new method was fast and convenient. In the end, the actor using the existing method was in chaos and then looked at the new system.
- **CONSEQUENCES:** The key for using Contrast pattern is to identify the right case where the benefit of the new system is overwhelming compared to the existing system.
- **SOURCE:** https://youtu.be/MTayd0kyY6Y
- **RELATED PATTERNS:** Conflict, Twist, Straightforward

[21] Homepage of iPraktikum 2016/2017: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/949

4.2.8 Straightforward pattern

- **CONTEXT:** Drama development for the story.
- **PROBLEM:** Not every project could be presented in a dramatic way. For example, some projects focus more on demonstrating the feasibility of a new technology, especially in industrial contexts, instead of highlighting the usefulness of the system in the user's everyday life.
- **SOLUTION:** As its name suggests, the Straightforward pattern presents the demo system in a plain and straightforward way, just going through all the functions step by step and shows the result to the audience. It is a special case for drama development because there is basically no dramatic element, e.g. conflict or twist, in the story.
- **CONSEQUENCES:** It is suggested to use the Straightforward pattern only when the Conflict, Twist, and Contrast patterns are all not fit in the story.
- **EXAMPLE:** The KneeHapp2 project from iPraktikum 2014/2015 [22] developed a sensor-based solution to rehabilitation. The project introduced a smart bandage with built-in sensors, which could be used to monitor the users' movements during the rehabilitation exercises, such as side hop, two-leg hop, and one-leg hop, etc. To demonstrate the new solution, the actors go through these functions: one actor with the bandage on the leg performed the movement, and the other actor watched the result recognized by the app on an iPad.
- **SOURCE:** https://youtu.be/E19A3_ZL2ig
- **RELATED PATTERNS:** Conflict, Twist, Contrast

[22] Homepage of iPraktikum 2014/2015: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/619

4.3 Software Theater Best Practices

In this section, we describe best practices for better preparing and performing demonstrations using Software Theater.

4.3.1 Setting up additional screens for synchronized illustration

Unlike traditional theatrical plays that focus on creating the atmosphere and developing the story, Software Theater focuses also on the details that are necessary for the audience to "touch and feel" the future system. The actors' performance according to the theater script reflects the *usage world*. It is suggested to show the user interface (the user-perceivable aspect of the *system world*) and relevant participatory objects (the *subject world*) whose status is managed by the system in the parallel. To this purpose, it is required to set up two screens on the stage, one for the user interface (via screen sharing) and the other for displaying the participatory objects in real time (via a moving camera). Figure 4.2 shows a recommended equipment setup for the stage of Software Theater. In this way, the three worlds could be shown to the audience simultaneously. The screen could also be used to project illustration slides about the demo system while the narrator is explaining the system.

4.3.2 Using video as an alternative to theater

There are situations where live demonstrations cannot be performed, for example, due to physical limitations of the stage or unavailability of certain demo components. In these cases, live demonstrations could be replaced by remote demonstrations via online video streaming or by playing recorded videos (called *Software Cinema* [Creighton2005, Creighton2006]) as an alternative solution. Video-based demonstrations could be considered as an "offline" version of Software Theater. As described in Section 4.1.1, the relationship between Software Cinema and Software Theater is like that of film and theater: films are not live performances and less interactive than theaters. However, compared to live demonstrations where "a gesture, once made, can never be made the

same way twice" [Monaco2013, p.60], videos are easier to be mocked up and, when necessary, use post-production techniques, such as *green-screen technology*, to achieve expected visual effects [Bruegge2008].

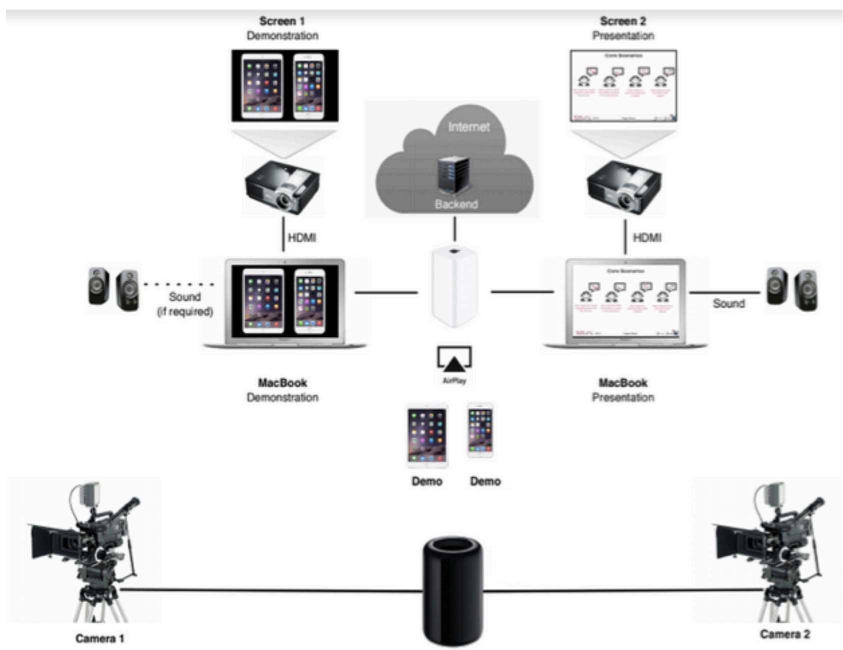


Figure 4.2 Recommended equipment setup for Software Theater

Besides, videos could also be used to record the performance of Software Theater for archival purposes. It is recommended to set up a dedicated camera for doing this. The archived videos could be useful at a later occasion. For example, at the customer acceptance test meeting, sometimes it is necessary to review the demonstration videos from the design review, because the video provides context information about the ongoing project as well as design decisions. As described by Bruegge [Bruegge2010, p6], "when acquiring knowledge and making decisions about the system or its application domain, software engineers also need to capture the context in which decisions were made and the rationale behind these decisions". Furthermore, Software Theater videos could also serve as training and reference purposes for similar projects in the future.

4.3.3 Illustrating negative sides as well

Software Theater is not only used to demonstrate the positive sides of the future system; instead, it could also be used to illustrate the negative sides of the future system. The purpose of Software Theater is to give the customers a faithful demonstration of the future system so that they could decide if they want to go productization or not. Showing constraints of the future system is essential for the decision. Besides, Software Theater could also be used to illustrate the negative sides of alternative solutions that were not adopted. By comparing both solutions to the same problem, the audience (including decision makers) could better understand the rationale behind the current design, especially when it is a matter of user experience because Software Theater could better present user's feelings toward the usability design.

4.3.4 Getting customers involved in creating demo scenarios

As depicted in Figure 3.2, demo scenarios are used as input for developing demo systems and for creating theater scripts. They are key artifacts created in the whole Demo Engineering process, and the customer's opinion is essential for the success of the project. Apart from showing concepts, user experience and technical feasibility of the future system, the demonstration should also address other aspects of concerns from the customer. Examples of important questions that should be acquired from customers include: what are scenarios for killer application of the future system, which are highly expected to be demonstrated; which parts are mocked and which parts are faithfully implemented; and so on.

Chapter 5
Software Theater Example and Evaluation

"So we believe that human factors may offer some of the best opportunities for innovation..."
-- David Kelly and Tom Kelly

In this chapter, we first describe the basic information about iPraktikum, a practical course at Technical University of Munich, which has been using Software Theater for exploratory projects with real customers. Then we take one of the projects as example and review each step of Software Theater in the real world setting. Finally we describe the evaluation of using Software Theater in iPraktikum.

5.1 The Practical Course of iPraktikum

At Technical University of Munich, the Department of Informatics has conducted practical courses for software engineering education. In recent years, this course is known as *iPraktikum* [23], which regularly receives about one hundred students (mostly majoring in computer science) every semester. In this practical course, participants are organized into teams and develop iOS-based applications. The projects come from real industrial customers with real-world requirements, ranging from mobile apps for end-users, proofs-of-concepts of IoT-based industrial applications, to feasibility studies of emerging

[23] More information can be found on the course website:
https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/1114

technologies. Unlike software projects aiming to implement well-defined requirements using matured technologies before the given deadline, these projects are of exploratory nature and in many times with only partially defined requirements or even open-ended requirements. For example, some customers would specify only system requirements (e.g., using the latest AR technologies) and request the project team to identify potential killer applications within their business domains (e.g., to propose next-generation concepts for their products or services).

According to our past experience in operating these practical courses, we limit the size of each team to around ten students, where everyone is supposed to participate in the development, and some are assigned additional functional roles, such as release management, usability design, and system modeling. Besides, each team has a dedicated *Project Leader* whose role is assumed by a PhD researcher in software engineering and a *Coach* whose role is filled by an experienced student that is chosen from developers of past courses. The project manager is the owner of the project and is responsible for ensuring the delivery of the project on time. The coach serves as a scrum master and takes care of the day-to-day problems, making sure that the project moves on along the agile process. To give necessary assistance to students taking additional roles (such as system modeling, release management, usability design, or being a coach), each role has an *Instructor Team*, which is made up of three PhD researchers in the relevant fields. Figure 5.1 illustrates the organizational structure for five projects from an iPraktikum.

The iPraktikum has a multi-project organization, where students are separated into teams, and each team works on a topic from a specific industrial customer [Bruegge2012, Bruegge2015]. The time frame for these projects is designed to match the length of one semester, including three milestones: Kick-off, Design Review, and Customer Acceptance Test (CAT). At each milestone, we organize a plenary meeting that expects all the stakeholders, including team members, coaches, project leaders, instructors, and customers, to participate.

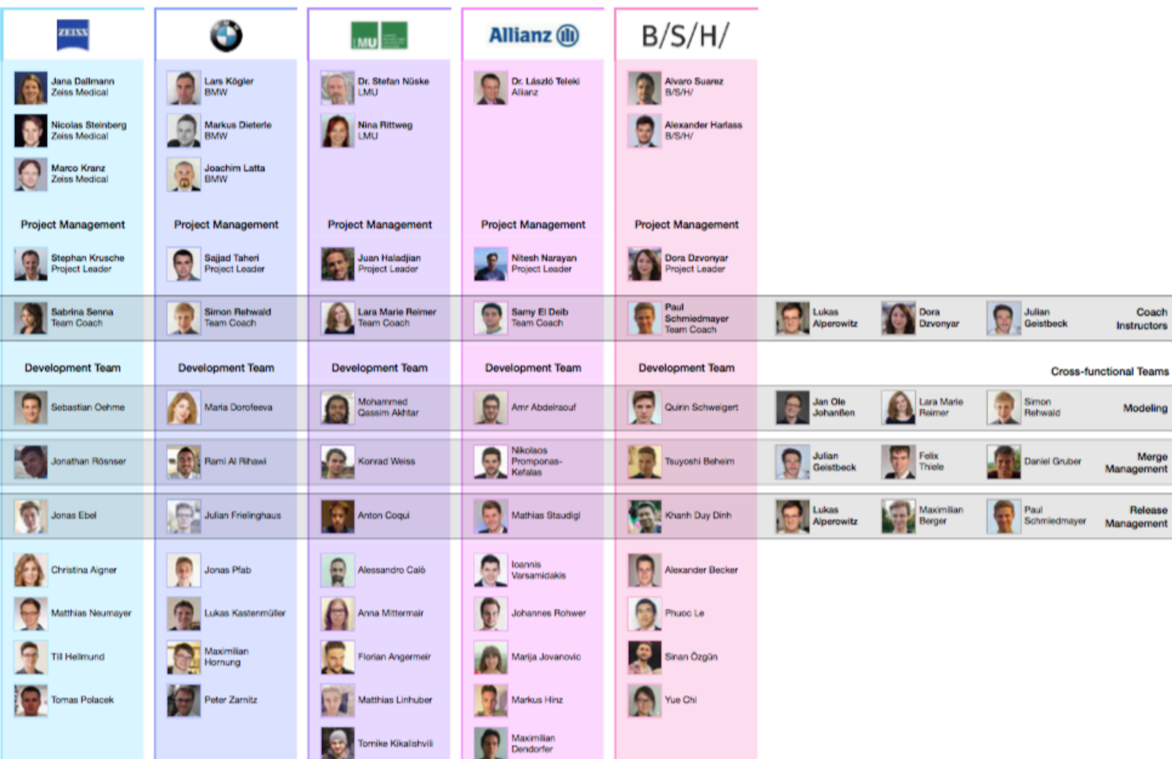


Figure 5.1 Example of the organizational chart for five projects from iPraktikum (from [Krusche2018])

- The *Kick-off meeting* takes place at the very beginning of the iPraktikum, where the industrial customers present their problems and requirements to all the students using slides, pictures, or videos. The main purpose of the kick-off meeting is for the students to know basically what the tasks are. After the meeting, the project teams should make clear and analyze the problems and requirements within two to three weeks. [Bruegge2012]. Then, in the following five to six weeks, they are supposed to finish requirements analysis and system design activities.
- The *Design Review meeting* is arranged eight weeks after the Kick-off meeting, where the project teams should present their results of requirement analysis and system design to the customers. While we highlight the importance of presenting technical architecture and design of the future system, we also encourage the

project teams to illustrate as-is scenarios and visionary scenarios using Software Theater and/or Trailer (i.e. video-based scenario, refer to Section 4.3.3). At this phase, the demonstration focuses on illustrating the main concept of the future system; therefore, the demo system is made up of more mock implementations than real implementations. To help the project teams to prepare for the demo systems and the demonstrations, the *Instructor Team* holds a course-wide lecture, where the instructors explain the Demo Engineering workflow and show videos of example demonstrations from the past. This lecture takes place three weeks before the Design Review so that the project teams would have enough time to apply them in their projects. Before the Design Review, the project teams are supposed to rehearse the demonstrations multiple times and iteratively adapt the interactions between the actors and the demo systems. A Dry-Run session without the presence of customers takes place one week before the official Design Review meeting, which gives each team a chance to practice the demonstration and receive feedback from the audience (other project teams). Performing Software Theater at this milestone is important because it gives the stakeholders a chance to evaluate the project in the middle and give realistic feedback to the project team. After refining the requirements based on the feedback, if necessary, the project teams move on to the implementation [Bruegge2015].

- After the Design Review, the project teams carry on the implementation for seven weeks and then present their final results (the demo systems) at the *Customer Acceptance Test (CAT) meeting*. At this phase, the demonstrations focus not only on the concepts but also on usability and technical feasibility. The demonstrations provide the customers with tangible feelings on the future system in terms of concepts, user experience, and technologies, so that they could decide to proceed to productization or not.

5.2 Example: The Zeyes Project [24]

To exemplify how Demo Engineering works from the demo-oriented development to the live demonstration using Software Theater, we describe a real world project for instance in the following. The project "Zeyes" is taken from iPraktikum 2016/2017 [25]. The industrial customer is Zeiss Meditec, a department of Zeiss, specializing in medical technologies. The goal of the project is to develop an app to support Zeiss Meditec's customers (e.g., surgeries) to do inventory management, including *stock taking* and *ordering new products*.

5.2.1 Preparation: from visionary scenarios to formalized scenarios

At the Kick-off meeting, the customer first introduced some basic information about Zeiss Meditec (including its history, business scope, and vision, etc.) and presented the general goal of the project (including the targeted users and intended situations for the future system, etc.). Next, since the topic of the project is for a vertical industrial domain (i.e., hospitals), the customer explained in more detail about the problems and challenges faced by end users (i.e., cataract surgeons) as well as their daily works, helping the team to better understand the context and capture the potential constraints. Then the customer defined main tasks for the team to solve (see Figure 5.2).

[24] This example was first described in Section 4 of [Krusche2018] and is adapted in this section.

[25] Homepage of iPraktikum 2016/2017: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/949

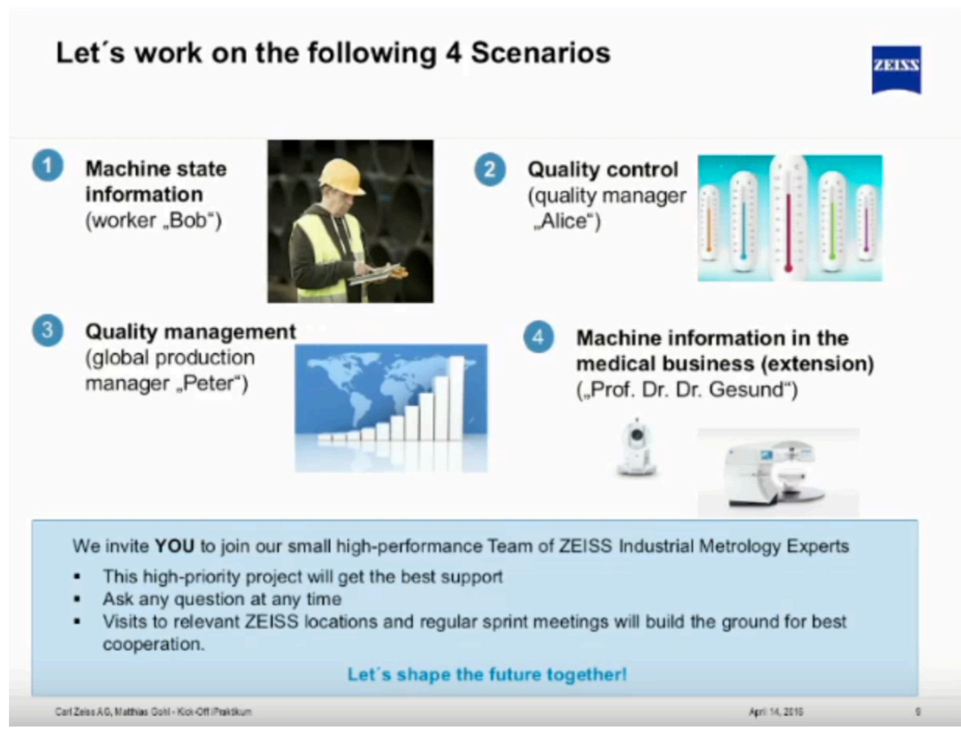


Figure 5.2 Main tasks defined by the customer

One of the visionary scenarios received from the customer was:

> *"The sales representative for Mexico is in charge of managing several customers. The sales process of consumables involves much more operational activities for the sales representative than the sales of devices. She needs to visit each customer regularly to initiate replenishment orders for the customer and to conduct stock checks. Both processes are quite manual today and reduce the time she can talk to the customer about new products and application related questions during her visit. The manual process introduces errors in stock taking.*
>
> *She already works with her smartphone for the customer account management, so she would appreciate if she could also use it for ordering and stock management. This would be desirable because mistakes in stock taking*

intro- duce organizational issues and costs, and every minute that she saves in this process can be used to have value added discussions with the customer." (From [Krusche2018])

As the project went on, after internal experiments and discussions with the customer, the team selected scenarios with innovative tryouts for the demonstration. Those scenarios were converted to formalized scenarios (a structured format for scenarios, including name, participating actors, flow of events, entry conditions, and exit conditions). Figure 5.3 shows one of the formalized scenarios from the project.

Scenario name	Perform stock taking	
Participating actors	Christina: Sales Representative, Matthias: Sales Representative	
Flow of events	**User steps**	**System steps**
	1) Tap on 'Mexican Medical Company' in the favorite customer list	
		2) Show customer locations
	3) Tap on the customer location 'Mexican Hospital 1'	
		4) Show the current stock item list
	5) Tap on 'Scan barcodes'	
		6) Activate camera
	7) Point camera at the barcode on each box	
		8) Identify the corresponding item and insert it into the scanned list. If an item is scanned twice, show an alert.
	9) Tap on 'Show Report'	
		10) Show the list of identified and missing items
	11) Uncheck 'Reorder items with report' and tap on 'Send report'	
		12) Show loading indicator and notification of sent report
Entry conditions	• The app is connected to the customer service hub and opened • The sales representative is logged in	
Exit conditions	• The correct report with all scanned items is sent to customer service hub	

Figure 5.3 A formalized scenario for the stock taking (from [Krusche2018])

5.2.2 Implementation: create the demo backlog and the demo system

The next step was to create the demo backlog based on the demo scenarios. Figure 5.4 depicts the subsystem decomposition that focuses on the components included in the demo backlog. The Zeyes app runs on the user's smartphone at the user's site (i.e., a

hospital), which is used to identify items by either scanning barcodes (using Smartphone Camera) or reading NFC tags (using NFC Reader) [26]. "It communicates with the Customer Service Hub to retrieve customer information, to submit reports, and to place orders." [Krusche2018] Since the team did not introduce any innovative tryout to the scenario of "submitting an order", this scenario was not necessary for the demonstration and was not included in the demo scenario. Therefore, the Order Manager and the Order Subsystem were not part of the demo system (greyed out in the figure). Besides, since NFC was not supported by iPhones at the time of the project, the team decided to mock up the NFC Reader component (marked in blue). Until then, the team had defined the scope of the implementation, and which parts went real implementations, which parts went mock implementations.

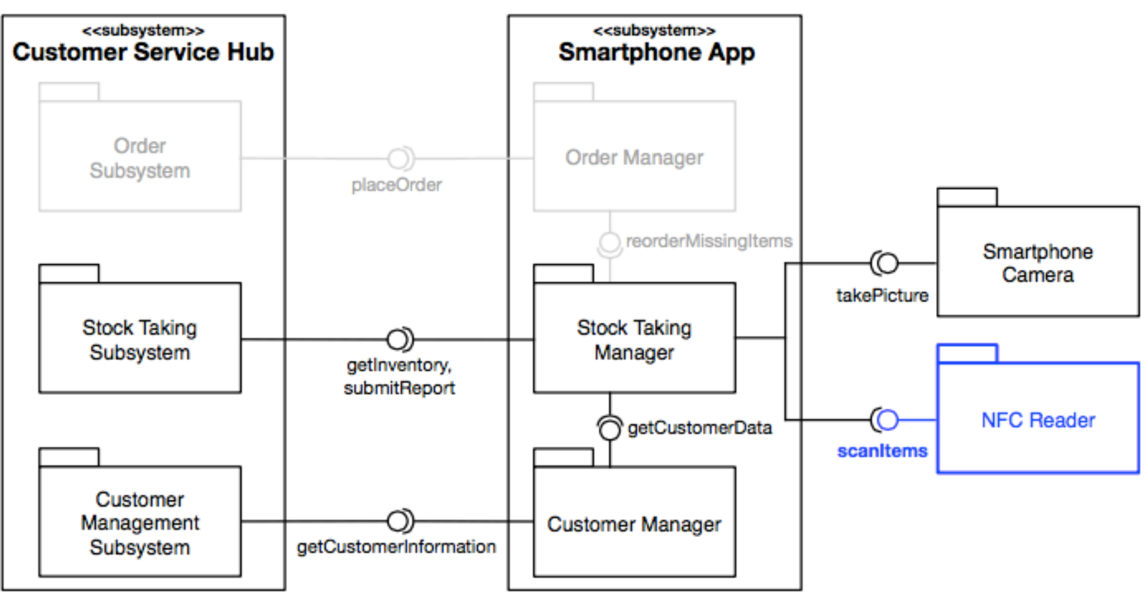


Figure 5.4 Participating components required by the demonstration (from [Krusche2018])

Next, the team further identified the participating objects and methods for implementation based on the selected demo components (see Figure 5.5). The blue parts

[26] NFC is short for Near Field Communication.

were components selected for mock implementations. For example, the NFC Connector would be indicated as available but always return the pre-defined data. The interface getInventory provided by the Stock Taking Subsystem was also selected for mock implementation because "the data of real server responses contained cryptic item names" [Krusche2018]; therefore, the team "changed the getInventory method in the Inventory object to return more understandable item names" for the demo system [Krusche2018].

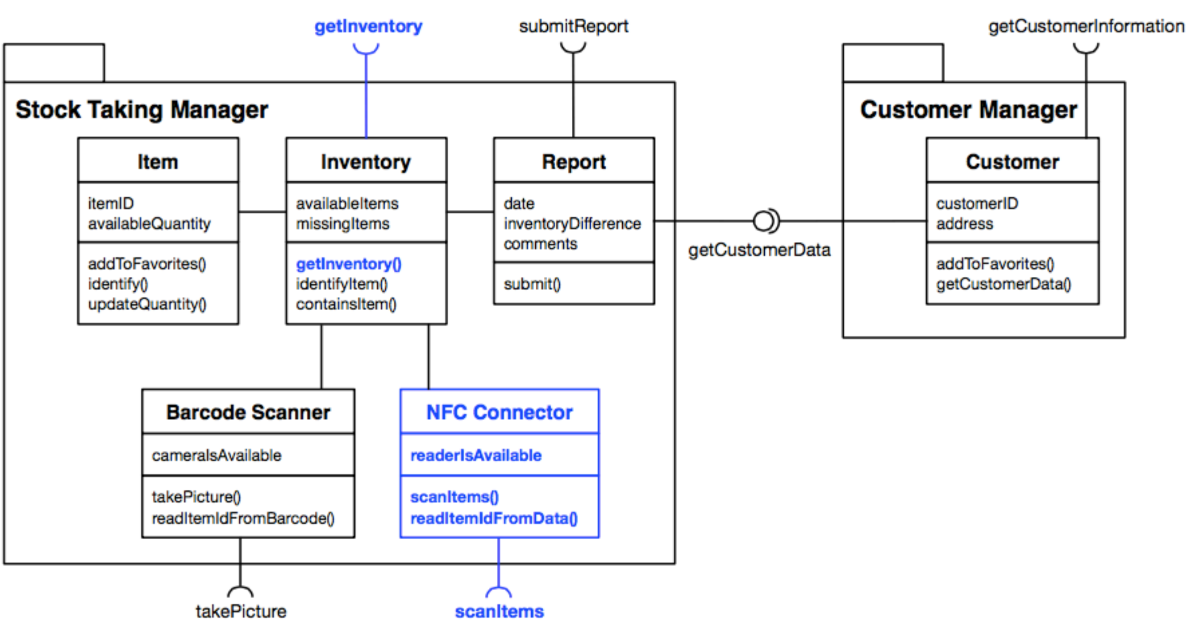


Figure 5.5 Participating objects and methods required for the demonstration (from [Krusche2018])

- [ ] Implement Barcode Scanner
- [ ] Implement getCustomerData interface
- [x] Fix error when scanning multiple items
- [ ] Mock NFC Reader
- [x] Print barcodes (props)
- [x] Get binders and stack of paper (props)
- [ ] Set up Mexican customer data
- [ ] Put stock in customer inventory
- [ ] Put boxes on shelf (props)
- [ ] …

Figure 5.6 Demo backlog with tasks to realize for the demonstration (from [Krusche2018])

"After identifying the participating methods and objects, the team created the demo backlog, including implementation tasks and organizational tasks." [Krusche2018] Figure 5.6 shows "an excerpt from the demo backlog for this demonstration, including action items for the implementations as well as for the preparation of props." [Krusche2018]

5.2.3 Presentation: performing the demonstration using Software Theater

When the demo system was implemented (including real implementations and mock implementations), the team started to prepare for the demonstration. First, they created the *theater script* to cover all the demo scenarios. They decided to set up a narrator leading the audience through the scene (refer to Section 5.1.2 for the *Narrator* pattern) and two actors (representing employees) performing the *stock taking* process. They also decided to compare as-is scenario and visionary scenario on the stage in the parallel: one actor (Christina) performing stock taking process manually (the traditional way) and the other actor (Matthias) performing the same task using the Zeyes app (the new way). Figure 5.7 shows an excerpt from the theater script, where you could see that Christina was instructed to show being frustrated while Matthias being more pleasant with the Zeyes app. Figure 5.8 shows one of the scenes of the live demonstration.

(A storage room with a big shelf with boxes with barcodes on them. CHRISTINA walks in with a stack of binders and lots of paper, almost tripping over her big pile. She puts it down on a desk and begins manual stock taking. MATTHIAS walks in with a smartphone in his hand.)

NARRATOR
These two are Christina and Matthias. They both are Sales representatives. They just entered a hospital to perform a cycle count in order to check which items are still there and which items need to be re-ordered. This is a customer service provided by sales representatives.

CHRISTINA
(fumbles around with the paper and notes thing down, having a hard time and getting confused and frustrated)

NARRATOR
On the one hand, Christina has to look up all the information she needs in her stack of paper. Matthias on the other hand, is using the newly developed App. Now, Matthias, can you please show us how the app works?

MATTHIAS
Yes, of course, I'll just hop right in. I open up the app and I can already see my favorite customers. These customers I visit regularly, and now I choose Mexican Medical Company (select Mexican Medical Company). Then I can see all the locations where they keep stock. We are in Mexican Hospital 1, let's select that (select the location).

Down here (point at the list), I can get a quick overview over every item that's supposed to be here. Let's check if this is true. I point the camera at the item (scan the barcodes of the boxes one by one) and it starts scanning.

CHRISTINA
(is obviously pretty stressed out and keeps dropping boxes. The storage room is a complete mess.)

MATTHIAS
One barcode is not here, we found a missing item! Let's klick on the report icon (click on the icon) and we can see which item is missing. The customer support should know that. We don't want to order the missing item (uncheck the option to order) so I just send the report.

CHRISTINA
I hate this, this is a complete disaster. I will never finish it in time.

MATTHIAS
(Walks over to Christina.) This is a horrible mess you have created.

CHRISTINA
It's such a mess. I will have to start all over again.

MATTHIAS
Well, I can help you with the new feature in the app. Check this out, we now have scanning via RFID in the app. So I can just press the button (press 'scan via RFID' button) and keep the device close to all items (wait for all items to appear)… and we're done. Every item was there, so I just send the report and your work is done.

CHRISTINA
Oh my God, this is exactly what I needed, then I'm not always the one to clean up the storage rooms of all the hospitals.

MATTHIAS
So, we're finished. Let's have dinner!
(both of them walk out together, smiling)

Figure 5.7 An example of the theater script (taken from [Krusche2018])

Figure 5.8 Software Theater: comparing stock taking manually vs. using Zeyes app (from [Krusche2018])

5.3 Evaluation [27]

The evaluation was conducted by surveying the participants of iPraktikum 2016/2017 [28]. In the following, we first introduce the design of the evaluation, and then describe the result and our analysis.

5.3.1 Evaluation design

We prepared a list of questions regarding the usage of Software Theater. These questions were sent to 91 participants (including team members and coaches) in the form of three-point Likert. The purpose is to evaluate the efficacy of using Software Theater in real-world projects. In the following, we list these questions as well as the relevant purposes.

- **Question 1: The other teams' Software Theater demos helped me to understand their projects better.**
  Purpose: Each team is supposed to be familiar with its own project but unfamiliar with projects developed by other teams. These questions are to evaluate if Software Theater is helpful for the audience without prior knowledge to understand the future system.
- **Question 2: Create a Software Theater demo gave us confidence about our system's usefulness.**
  Purpose: Presenting a new system with merely mock implementations is only at the concept level. Developers would feel more confident if the potential uncertainties in the system could be verified with real implementations. Besides, demonstrating the new system in representative scenarios create more confidence in its usefulness.

[27] The evaluation is based on a survey that was first described in Section 6 of [Krusche2018].

[28] Homepage: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/949

- **Question 3: I want to use Software Theater to create demonstrations in future projects**
  Purpose: This question is to evaluate if people are willing to use Software Theater in the future, which reflects their overall satisfaction on it.
- **Question 4: Creating a Software Theater demo helped my team to understand the project requirements.**
  Purpose: It is assumed that demonstrating the system in the context helps develop empathy and visual sense about the usage of the system. This question is to evaluate that assumption from the team member's perspective.
- **Question 5: Creating a Software Theater demo helped my team to communicate with the customer.**
  Purpose: In our opinion, Software Theater is yet another informal model for communication. This question is to evaluate if Software Theater helps the communication between developers and customers.
- **Question 6: I would have preferred to prepare a demo without Software Theater.**
  Purpose: This question is introduced as a reversed worded item to reduce response bias [Paulhus1991].

5.3.2 Evaluation results

We received 80 responses from 11 teams in total, among which 70 were team members and 10 were team coaches, and the overall response rate was 88 %.

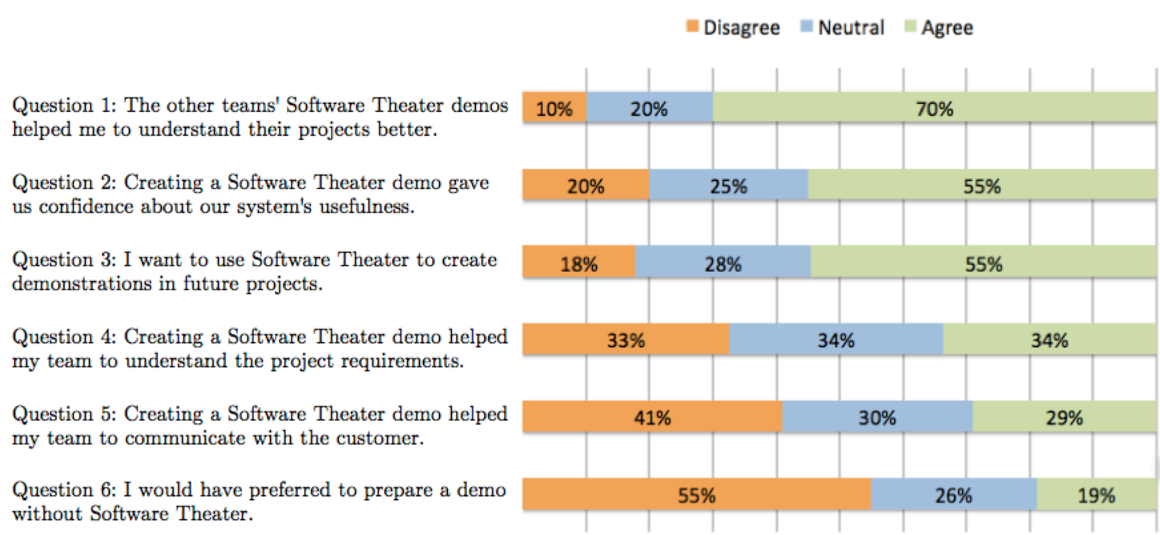


Figure 5.9 Results of the 3-point Likert survey (Adapted from [Krusche2018])

The results of the three-point Likert survey are shown in Figure 5.9. In the following, we go through every question and analyze the results.

- **Question 1: The other teams' Software Theater demos helped me to understand their projects better.**

  Analysis: It is a strong positive result that 70% of the participants agreed with the assumption. This is as expected, and we are confident about this result since the intuitiveness is the nature of theatrical plays.

- **Question 2: Creating a Software Theater demo gave us confidence about our system's usefulness.**

  Analysis: 55% of the participants agreed with the assumption. It is a positive result, but a bit lower than our expectations. However, considering that 25% of participants chose neutral, which means that only 20% were negative, the result is acceptable. In our opinion, the criteria of "being confidence" (in this question) are more subjective than "understandable" (in the first question), which might be a reason that participants tended to choose "Neutral" instead of "Agree".

- **Question 3: I want to use Software Theater to create demonstrations in future projects.**
  Analysis: 55% of the participants agreed that they would use Software Theater in future projects, and 28% stayed neutral. We are satisfied with this positive result because the difference in the nature of projects may influence their overall opinions toward Software Theater. For example, if a project were in lack of exploratory nature, then it would be harder for the team members to experience the benefits of Software Theater.

- **Question 4: Creating a Software Theater demo helped my team to understand the project requirements.**
  Analysis: The result shows that the opinions were divided concerning this question. Each opinion (agree, neutral, and disagree) accounts for about 33% proportion. After analysis, we believe that the subjective nature of this question influenced some participants to choose neutral; besides, for projects whose requirements were simple and clear, they would have the tendency of not attributing the understanding of project requirements to Software Theater.

- **Question 5: Creating a Software Theater demo helped my team to communicate with the customer.**
  Analysis: 29% of the participants agreed, 30% were neutral, and 41% disagreed. We attribute the result to the same reasons as in Question 4. They disagreed more because "communicating with the customer" is more difficult for the participants to measure than "understanding the project requirements".

- **Question 6: I would have preferred to prepare a demo without Software Theater.**
  Analysis: Only 19% of the participants would have preferred not to use Software Theater in their projects; 55% disagreed, and 26% were neutral. These results meet our expectations because not every project was of exploratory nature, and different team members have their individual differences toward the usage of Software Theater. These biases contributed to the negative and neutral opinions.

Table 5.1 Correlations of the Hypotheses and Questions

Hypothesis	Relevant Questions
H1) Software Theater is suitable for presenting visionary systems in the context of exploratory projects	Question 3
H2) Software Theater is more intuitive and engaging than textual descriptions for users to "see" (i.e., understand and envisage) the future system.	Question 1, 4, 5
H3) Software Theater creates more faithfulness and confidence for the evaluation of the future system	Question 2

According to the questions analyzed above, we have the following evaluation of our hypotheses and the overall results show that the hypotheses are supported.

- **Hypothesis 1) Software Theater is suitable for presenting visionary systems in the context of exploratory projects.**
  Analysis: This hypothesis is strongly supported by the results of Question 3. Projects with exploratory nature (53%) chose to use Software Theater in the future; as a comparison, a small number of projects (18%) that are less exploratory would not use it in the future.

- **Hypothesis 2) Software Theater is more intuitive and engaging than textual descriptions for users to "see" (i.e., understand and envisage) the future system.**
  Analysis: The results of Question 1 provide strong evidence that Software Theater helps users to understand the project better (agreed by 70% of the participants). In addition, the results of Question 4 show that Software Theater, in some projects, help to understand the requirements, which will lead to the development of the future system. The results of Question 5 show that Software Theater, in some projects, is more efficient for the team to communication with the customer.

- **Hypothesis 3) Software Theater creates more faithfulness and confidence for the evaluation of the future system.**

  Analysis: The results of Question 2 verify that Software Theater gives the team members confidence about the system's usefulness, which was agreed by the majority of the participants (55%).

5.3.3 Threats to validity

One limitation of the evaluation is that we did not set up a control group because we did not have two teams working on exactly the same topic in our projects. Another threat is that the survey based on Likert scales may be subject to distortion [Garland 1991]. For instance, respondents may have avoided using extreme response categories (central tendency bias), and they may tend to agree with what we have stated in the question (acquiescence bias). They may also suffer from social desirability bias. For example, they might not want to give negative feedback because they try to portray themselves or the practical course in a more favorable manner. To mitigate these threats, we did the survey in an anonymous way and prevented multiple responses from the same person. Another threat to the validity of the evaluation is that most students (developers in project teams) were excited about carrying out projects with real customers, which might give them the tendency to give positive feedback. To address this potential threat, we deliberately added an item in reversed wording to the question list, such as Question 6.

Chapter 6
Conclusion

"The advance of technology is based on making it fit in so that you don't really even notice it, so it's part of everyday life."
--- Bill Gates

This dissertation has described how to use Demo Engineering with Software Theater for exploratory projects. Adopting new technologies and new systems is not always easy and straightforward for the users because they will have to renovate their current practices that have been used in their everyday lives, and they might worry about the uncertainties resulting from the new concepts, new experience, and new enabling technologies. As illustrated in this dissertation, Software Theater, as an intuitive, lifelike, and sympathetic means of communications, fills these gaps and enables the users to "touch and feel" the new system. In the following, we elaborate on the contributions of this dissertation and propose directions for future works.

6.1 Contributions

The major contributions of this dissertation is the systematic description of *Demo Engineering and Software Theater*, including the concept, the principles, the workflow as well as the patterns and best practices. We have explained why Demo Engineering and Software Theater are introduced, what they are, as well as when and how they should be used.

- **Why.** Exploratory projects are to explore new possibilities by trying out new concepts, new experience, and new enabling technologies. They bring not only

novelty but also uncertainties. Demo Engineering and Software Theater are to address these challenges of exploratory projects (see Section 3.1).

- **What.** Software Theater focuses on demonstrating systems based on theater scripts and presenting them in a theatrical way. Software Theater provides a way to present these demo systems in a lifelike setting so that the applicability of new concepts, the usability of new user experience, and the feasibility of new technologies could be evaluated reliably (see Section 3.2).

- **When.** Software Theater is particularly used review meetings, such as sprint review, design reviews and final deliveries. The demonstration at the design review focuses more on the illustration of concepts and user experience while the demonstration at the deliveries focuses on system acceptance testing (see Section 3.3.3).

- **How.** The workflow of Demo Engineering is divided into three activities: preparation (from visionary scenarios to formalized scenarios), implementation (creating demo backlog and the demo system), and presentation (performing the demonstration using Software Theater) (see Section 3.3). Besides, we also described patterns and best practices for using Software Theater (see Section 4).

Another contribution of this dissertation is that we have reviewed research works on scenario-based design, based on which we extended the existing understanding of scenarios and made further analysis.

- We identified three-model conceptualization as a useful analysis tool for scenarios describing ubiquitous applications. Unlike traditional desktop applications, these ubiquitous applications involve not only the system world and usage world but also the subject world (see Section 2.1.2).

- We revealed the dual perspectives of scenarios and promoted the distinction of *scenarios as artifacts* and *scenario creation as a process.* This distinction brings us insight into the properties and usages of scenarios, and enables us to use appropriate Forms of scenarios for different Purposes (see Section 2.3).

- We used *yin and yang* as an analysis tool for analyzing the relationship between user models and systems models, and promoted the co-development of user models and system models, which forms one of the principles behind Demo Engineering (see Section 3.4.1).

6.2 Future Works

This dissertation forms a basis for using Software Theater, and we have used it for real-world projects already. However, if relevant supporting tools were available, Software Theater would become more popular and easier to be adopted. In the following, we point out two directions for developing such Software Theater tools.

6.2.1 Theater script editor

Creating theater scripts is a requisite step for the stage performance of Software Theater. There are screenplay writing tools on the market, such as Celtx, Final Draft, etc. [Batty2014]. However, as described in Section 4.1, the performances in Software Theater are short in terms of duration and employ only limited dramatic techniques, therefore, those tools for professional screenplay writers are too heavyweight for use in Software Theater. What is required is a theater script writer that just fits for Software Theater. In the light of the characteristics of Software Theater, the theater script writer could provide predefined templates for typical structures of story (guiding beginner users to have a quick start with theater script writing), hints for using Software Theater best practices and patterns (helping delivering better performances), correlation between demo scenarios and theater script (ensuring that the demo scenarios selected for demonstration are fully covered in the theater script). For example, to create a theater script for an ubiquitous application, the theater script editor could guide the user to consider the flow of events in three worlds: the system world shows the user interface of the system (e.g., mirroring the screen of a mobile phone to the projection screen at the forefront of the stage); the usage world illustrates the interaction between the user and

the system as well as its context (e.g., the actor on the stage operates on the mobile phone in a specific situation); the subject world shows the status changes of the objects managed by the system (e.g., using a second camera to live stream a smart home device to the second projection screen on the stage).

6.2.2 Software for creating scenario videos based on virtual worlds

We experimented with creating video-based scenarios using virtual world technologies in 2012 [Xu2012]. We have verified the technical feasibility of creating videos by programming using the OpenSim platform. As a prototype project, we created videos using existing 3D models (e.g. actors and objects). The massive usage of the approach relies on the creation of customized 3D models on the user's own, which requires 3D reconstruction equipment. Such equipment was still expensive in 2012; however, with the popularity and mass production of depth sensors in recent years, the price of 3D reconstruction devices have dropped substantially to a level that is affordable by ordinary users today. Therefore, now it is a good timing to further explore the possibility of building the software for creating video-based scenarios using today's hardware and virtual world platforms based on the previous prototyped work [Xu2012]. This software will provide another option for creating scenario videos in a programmable way.

Appendix
Examples of Theater Scripts

Example 1: ZeissMed BrainGame Project [29]

[The young pathologist comes into the library of the pathology with a stack of heavy books, that are supposed to prepare him for his work at the Pathology. PROPS: with the mobile phone in the pocket]

NARRATOR: This is James on his first day of work at the pathology.

[James has heard about the new Convivo technology they use here, and has therefore brought with him all the books he could possibly
find on the subject, hoping to impress his new boss.
The old pathologist is on his phone just giving him a glance - then looking really surprised...]

OP: Hi James, good to see you Would you mind telling me what you are doing with all these books?

[The young pathologist says proudly]

YP: I am preparing for my internship sir.

[The old pathologist shakes his head and beard]

OP: Did no one tell you we make diagnoses based on Convivo images here.
And Zeiss has come out with this app called BrainGame that teaches you how to diagnose CONVIVO images.

[The young pathologist in a sceptic voice]

YP: Soooo I don't need all these heavy books?

OP: No Go ahead and download the BrainGame App...

---

[29] This project is from iPraktikum 2019/2020. More details could be found at:
https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/1063

YP: OK I downloaded the app...

[The young pathologist connects the phone to the beamer and opens the BrainGame App for the first time. The PROJECTION SCREEN shows the mirroring screen of the phone.]

OP: See this is the Lessons Area where you have multiple Lessons with tutorials and questions where you can learn to diagnose Convivo images step by step. But lets first set up your profile...

[The young pathologist tabs on the profile image in the tab bar and is asked to setup his profile.]

OP: Oh I forgot you don't have a ZeissId jet so you can't log in...But you can start studying without it and the app will still track your progress (that you can see here), you can also set learning reminders (down here) and visit your bookmarked tutorials and questions (here).

OP: Have fun studying!

YP: Thanks I will get straight to it - wait what is this area good for?

[The young pathologist clicks on the practice area.]

OP: This is the Practice Area. Here you will be asked Questions randomly. You can also apply filters if you want to practice something a little more specific.

[The young pathologist clicks through the practice area.]

OP: For the beginning let me apply a filter that gives you only very simple yes/no questions.

[The old pathologist applies the filters and filters for the QuestionType YesNo, closes the filter and hands the phone back to the young pathologist.

The young pathologist looks at the question for a second - then smiles]

YP: I think I know the answer to this one, I think this shows a glioma... It looks like I was correct

[The young pathologist looks at the old pathologist hoping for acknowledgment]

OP: Yeah but you still have a lot of practice ahead of you till you can pass the final test - good look!

[The young pathologist looks confused but the old pathologist has gotten up and is about to leave]

YP: What final test???

[The old pathologist turns around and says in a mysterious voice]

OP: You will see when you have completed all the Lessons in the Lessons Area

[The old pathologist leaves the scene; the presentation is carried on...The PROJECTION SCREEN switches to the slides of the presentation. The MAIN CAMERA moves the focus from James to the presenter.

...several few minutes later, after the presentation. The MAIN CAMERA moves the focus to James.

The young pathologist jumps out of his chair and talks to himself]

YP: Yes I have completed the final Lesson, lets see if I have unlocked a Test...

[The young pathologist connects the iPhone to the beamer and is in the LessonsArea of the App. The PROJECTION SCREEN switches back to the mirroring screen of James' mobile phone.]

YP: Oh yes there it is - I am so ready to do this - I have studied so hard the last 3 minutes..... Ahm 3 Weeks

[The young pathologist clicks on the test and starts reading the test description]

YP: [reads out test description]

[The old pathologist enters the scene interrupting the young pathologist. PROPS: a carton board printing ZeissID and Password on it.]

OP: Hey James, how is it going.

[The young pathologist acts really excited]

YP: I am just about to start the final test

[The young pathologist clicks on start test and is asked for his ZeissID and Password]

YP: Oh I need my ZeissID and Password

OP: Ah I think I have it with me.

[The old pathologist shows to James the ZeissID and Password on the carton board]

OP: Here you go.

[The young pathologist enters his ZeissID and Password receives the message that his ZeissID and Password are correct and starts the Test]

OP: Also I would upload a profile picture if you want to have one on your certificate!

YP: Oh I think I have just the right one...

[The young pathologist opens the library selects the image and clicks on ok]

YP: Nice, I have successfully logged in!

[The young pathologist starts the test]

OP: Good luck!

YP: Thank you

[The young pathologist starts takes the test, the app shows a 29 minutes later screen]

NARRATOR: 29 minutes later James is answering his final question

[The young pathologist answers the last question and gets his test result]

YP: Yes I have successfully completed the test, now I only have to click the "Request Certificate" button and I will hopefully receive the certificate later so that I will be officially allowed to make diagnoses based on Convivo images.

[The young pathologist sends an email to Zeiss and gets a notification that the email was successfully sent. Immediately, somebody walk toward James, PROPS: with the certificate in hand, and hand it over to James]

NARRATOR: Congratulations James!

Example 2: The Quartett Mobile Project [30]

[PROPS: restart Wiremock server; Tobias sends commands to e-tron via myAudi to wake it up]

[Andrea leaves her car and walks towards Bob's place]

[Bob opens door and they welcome each other]

B: Did you find a close parking space? The situation is crazy here lately.

A: Not at all, I had to park 2 streets away. But I got here with my new e-tron :D

B: Oh that's cool!

A: Wait, let me check if my car is locked.

[A pulls out iPhone, asks Siri "Hey Siri, did I lock my car?"]

[PROJECTION SCREEN: livestream from iPad shows car, if it is outside lecture hall]

B: I always forget that as well…

[Siri responds that the car is unlocked, A confirms to lock car]

B: Luckily you've got this app! Actually, can you show me how this works?

[Car is locked, lights flash]

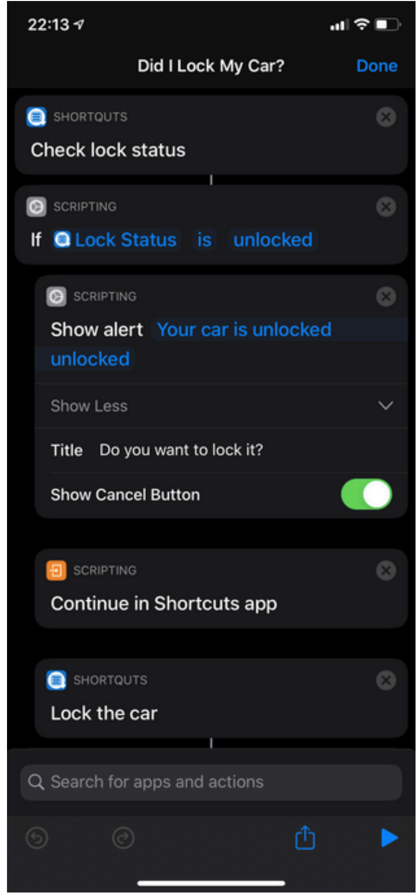
A: Sure!

[Both sit down]

A: This is the shortcut I just used.

[A opens Apple Shortcuts, shows conditionals, maybe explain a little]

[30] This project is from iPraktikum 2019/2020. More details could be found at: https://ase.in.tum.de/lehrstuhl_1/component/content/article/106-teaching/1063



A: We can also get the car's location, that's going to be handy later when I need to find my car again. And I can even send an address to the in-car navigation!

[Opens "Take me home" shortcut, briefly explains the two actions]

B: Hm, sending a message to your partner would be cool, right?

A: Yeah, let's add that!

[A adds message block at end of shortcut]

A: I think I've gotta go now.

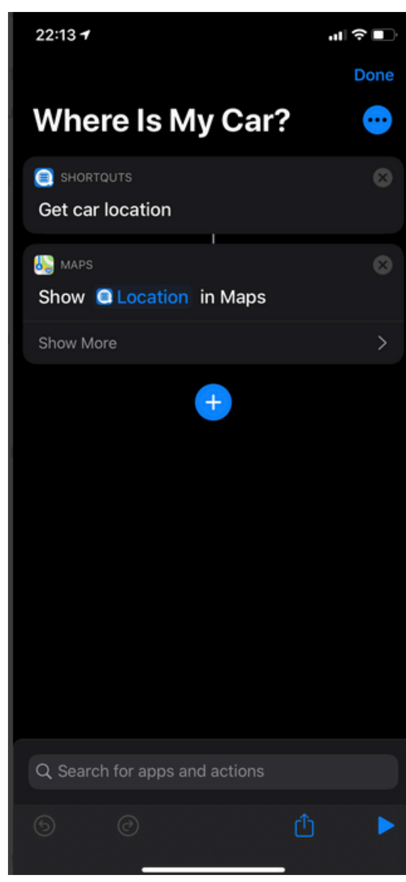
[They get up]

B: Anyway, it was nice to see you again!

A: Absolutely. Bye!

B: Bye!

[Andrea leaves, starts "Where is my car" via Siri]



[Maps opens]

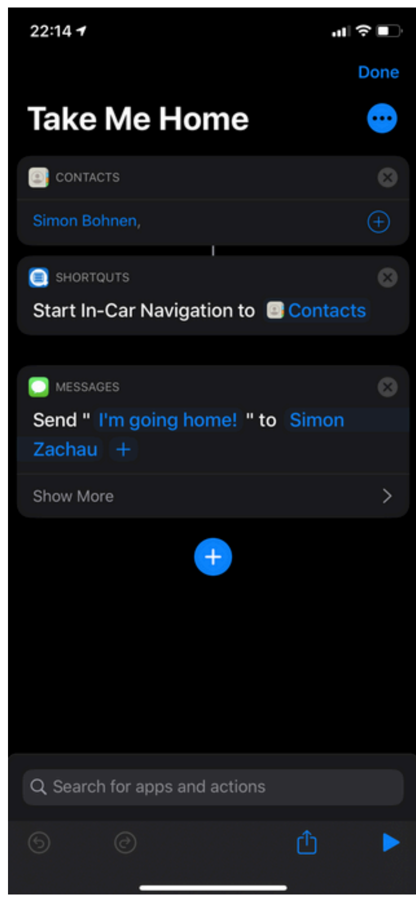
A: Nice, here I got the directions to my car.

Narrator:

[Communication with car is complicated and response times vary a lot, phone -> server, server -> car via sms, car wakes up, retrieves task from server, executes task, send response to server, phone requests task status]

A: Let's also send my home address to the in-car navigation.

[starts "Take me home" shortcut]



[finds car after some time]

Ah, there it is, let's see if my address is already put in.

[home address is shown via stream with iPad]

This app is really handy!

Bibliography

[Abbott1983] Russell J. Abbott. Program Design by Informal English Descriptions. *Communications of the ACM*. Volume 26, Issue 11, 1983.

[Achour1998] Camille Ben Achour. Writing and Correcting Textual Scenarios for System Design. *Proceedings of the 28th Natural Language and Information System Workshop*. 1998.

[Alexander1977] Christopher Alexander. A pattern language: towns, buildings, construction. Oxford University Press. 1977.

[Anton1998] Annie I. Antón and Colin Potts. A representational framework for scenarios of system use. *Requirements Engineering*. Volume 3, Issue 3-4, 1998.

[Batty2014] Craig Batty. Show Me Your Slugune and I'll Let You Have the Firstlook: Some Thoughts on Today's Digital Screenwriting Tools and Aprs. *Media International Australia. Volume 153, Issue 1, 2014*.

[Beck1989] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. *Proceedings on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 1989.

[Beck1999] Kent Beck. Change with Extreme Programming. *IEEE Computer*. Volume 10, Issue 32, October 1999.

[Boehm2000] Barry Boehm. Requirements that Handle IKIWISI, COTS, and Rapid Change. *Computer*. July, 2000.

[Ben1999] Camille Ben Achour and Carine Souveyet. Bridging the Gap between Users and Requirements Engineering. International journel of computer systems science & engineering (Special issue on object-oriented informationsystems). 1999.

[Benyon2002] David Benyon and Catriona Macaulay. Scenarios and the HCI-SE design problem. *Interacting with computers*. Volume 14, 2002.

[Bushmann1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture*. John Wiley&Sons. 1996.

[Brajnik2014] Giorgio Brajnik and Cristina Giachin. Using sketches and storyboards to assess impact of age difference in user experience. *International Journal of Human-Computer Studies*. Volume 72, Issue 6, 2014.

[Bruegge2008] Bernd Bruegge, Harald Stangl and Maximilian Reiß. An experiment in teaching innovation in software engineering. *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA),* 2008.

[Bruegge2010] Bernd Bruegge and Allen H. Dutoit. Object-Oriented Software Engineering: Using UML, Patterns, and Java (Third Edition). Prentice Hall. 2010.

[Bruegge2012] Bernd Bruegge, Stephan Krusche and Martin Wagner. Teaching Tornado: From Communication Models to Releases. *Proceedings of the 8th edition of the Educators' Symposium*. 2012.

[Bruegge2015] Bernd Bruegge, Stephan Krusche and Lukas Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education.* Volume 15, Issue 4, 2015.

[Caldwell2017] Craig Caldwell. Story Structure and Development: A Guide for Animators, VFX Artists, Game Designers, and Virtual Reality. CRC Press. 2017.

[Cao2008] Lan Cao and Balasubramaniam Ramesh. Agile requirements engineering practices: An empirical study. *IEEE Software*. January/Febuary, 2008.

[Carlson1986] Marvin Carlson. Psychic polyphony. *Journal of Dramatic Theory and Criticism*. September, 1986.

[Carroll1990] John M. Carroll and Mary Beth Rosson. Human-Computer Interaction Scenarios as a Design Representation. *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences.* 1990.

[Carroll1995] John M. Carroll. Scenario-based design: envisioning work and technology in system development. John Wiley and Sons. 1995.

[Carroll1998] John M. Carroll, Mary Beth Rosson, Juergen Koenemann. Requirements Development in Scenario-Based Design. *IEEE Transactions on Software Engineering.* Volume 24, Issue 12, 1998.

[Carroll2000] John M. Carroll. Making Use: Scenario-Based Design of Human-Computer Interactions. The MIT Press. 2000.

[Chan1963] Wing-Tsit Chan. *A Source Book in Chinese Philosophy*. Princeton University Press. 1963.

[Cleary1992] Thomas Cleary, trans. *I Ching - The Book of Change*. Shambhala Publications. 1992.

[Cohn2004] Mike Cohn. User Stories Applied: For Agile Software Development. Addison-Wesley. 2004.

[Control2008] Project Portfolio Control/Portfolio Management Performance/Different Contexts: Project Portfolio Control and Portfolio. *Project Management Journal*. Volume 39, Issue 2, 2008.

[Cooper2004] Alan Cooper. *The inmates are running the asylum.* Sams-Pearson Education. 2004.

[Cooper2005] Pat Cooper and Ken Dancyger. *Writing the Short Film (Third Edition)*. Elsevier Focal Press. 2005.

[Cooper2007] Alan Cooper, Robert Reimann and David Cronin. *About Face 3.0: The essentials of interaction design.* Wiley Publishing. 2007.

[Creighton2005] Oliver Creighton. Software Cinema: Employing Digital Video in Requirements Engineering. PhD Thesis. Technical University of Munich. 2005.

[Creighton2006] Oliver Creighton, Martin Ott and Bernd Bruegge. Software Cinema-Video-based Requirements Engineering. *Proceedings of the 14th IEEE International Requirements Engineering Conference*. 2006.

[Dabholkar2013] Vinay Dabholkar and Rishikesha T. Krishnan. *8 Steps to innovation: Going from Jugaad to Excellence.* HarperCollins Publishers. 2013.

[Fang2012] Tony Fang. Yin Yang: A New Perspective on Culture. *Management and Organization Review.* Volume 8, Issue 1, 2012.

[Freytag1898] Gustav Freytag. Technique of the Drama (2nd Edition). Translated by Elias J. MacEwan. Scott Foreman and Company. 1898.

[Gamma1996] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Software.* Addison-Wesley. 1996.

[Go2014] Kentaro Go and John M. Carroll. The blind men and the elephant: Views of scenario-based system design. *Interactions*. November, 2004.

[Goldsmith2004] Robin F. Goldsmith. Discovering Real Business Requirements for Software Project Success. Artech House. 2004.

[Goodwin2011] Kim Goodwin. *Interview: Kim Goodwin - Developing Effective Scenarios.* Retrieved May 19, 2015 from: http://www.uie.com/brainsparks/2011/08/05/kim-goodwin-developing-effective-scenarios/.

[Goodwin2011a] Kim Goodwin. *Designing with Scenarios: Putting Personas to Work*. Retrieved Jan 16, 2020 from: https://uie.fm/shows/spoolcast/kim-goodwin-designing-with-scenarios-putting-personas-to-work.

[Grudin2006] Jonathan Grudin. Why Personas Work: The Psychological Evidence. *The persona lifecycle: Keeping people in mind throughout the product design.* Morgan Kaufmann. 2006.

[Gudjonsdottir2010] Rosa Gudjonsdottir. *Personas and Scenarios in Use*. PhD Thesis. KTH Royal Institute of Technology. 2010.

[Haumer1998] Peter Haumer. Requirements Elicitation and Validation with Real World Scenes. *IEEE Transactions on Software Engineering*. Volume 24, Issue 12, 1998.

[Campbell2004] Joseph Campbell. The Hero With a Thousand Faces (Third Edition). Princeton University Press. 2004.

[Jacobson1992] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley. 1992.

[Jarke1992] Matthias Jarke, John Mylopoulos, Joachim W Schmidt and Yannis Vassiliou. DAIDA: An Environment for Evolving Information Systems. *ACM Transactions on Information Systems*. Volume 10, Issue 1, 1992.

[Jarke1993] Matthias Jarke and Klaus Pohl. Establishing visions in context: towards a model of requirements processes. *Proceedings of the 14th International Conference on Information Systems.* 1993.

[Jarke1998] Matthias Jarke, X. Tung Bui and John M. Carroll. Scenario Management: An Interdisciplinary Approach. *Requirements Engineering*. Volume 3, Issue 3–4, 1998.

[Jarke1999] Matthias Jarke. CREWS: Towards Systematic Usage of Scenarios , Use Cases and Scenes. CREWS Report 99-02. 1999.

[Jarke2011] Matthias Jarke, Pericles Loucopoulos, Kalle Lyytinen, John Mylopoulos and William Robinson. The brave new world of design requirements. *Information Systems*. Volume 36, Issue 7, 2011.

[Jiang2010] Hao Jiang, John M. Carroll and Roderick Lee. Extending the Task-Artifact Framework with Organizational Learning. *Knowledge and Process Management*. Volume 17, Issue 1, 2010.

[Kohavi2013] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu and Nils Pohlmann. Online Controlled Experiments at Large Scale. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2013.

[Krusche2016] Stephan Krusche. *Rugby - A Process Model for Continuous Software Engineering*. PhD Thesis. Technical University of Munich. 2016.

[Krusche2018] Stephan Krusche, Dora Dzvonyar, Han Xu and Bernd Bruegge. Software Theater — Teaching Demo Oriented Prototyping. *ACM Transactions on Computing Education.* Volume 1, Issue 1, January 2018.

[Laurel2014] Brenda Laurel. *Computers as Theatre (Second Edition)*. Addison-Wesley Publishing. 2014.

[Lenfle2008] Sylvain Lenfle. Exploration and project management. *International Journal of Project Management.* Volumn 26, Issue 5, 2008.

[Leffingwell1999] Dean Leffingwell and Don Widrig. *Managing Software Requirements.* Addison-Wesley. 1999.

[Long2009] Frank Long. Real or Imaginary: The Effectiveness of Using Peronas in Product Design. *Proceedings of the Irish Ergonomics Society annual conference.* 2009.

[LouisAlain2019] LouisAlain. "*Theater", Wikipedia.* Retrieved March 29, 2019 from: https://en.wikipedia.org/w/index.php?title=Theatre&oldid=885842072 [02.03.2019].

[Luecke2003] Richard Luecke and Ralph Katz. *Managing creativity and innovation: practical strategies to encourage creativity*. Harvard Business School Press. 2003.

[Mackinnon2000] Tim Mackinnon, Steve Freeman and Philip Craig. Endo-Testing: Unit Testing with Mock Objects. *Extreme programming examined.* Addison-Wesley Longman Publishing. 2001.

[Madsen1993] Kim Halskov Madsen and Peter H. Aiken. Experiences using cooperative interactive storyboard prototyping. *Communications of the ACM.* Volume 36, Issue 6, 1993.

[Mahaux2008] Martin Mahaux and Neil Maiden. Theater improvisers know the requirements game. *IEEE Software*. Volume 25, Issue 5, 2008.

[Mahaux2010] Martin Mahaux, Patrick Heymans and Neil Maiden. Making it all up: Getting in on the Act to Improvise Creative Requirements. *Proceedings of the 18th IEEE International Requirements Engineering Conference*. 2010.

[Maiden1998] Neil Maiden. CREWS-SAVRE: Scenarios for acquiring and validating requirements. *Automated Software Engineering*. Volume 5, Issue 4, 1998.

[Martin1996] Robert Cecil Martin. *The Dependency Inversion Principle*. C++ Report. 1996. Retrieved May 4, 2015 from: https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf

[Monaco2013] James Monaco. *How to Read a Film: Movies, Media, and Beyond (4rd Edition)*. Harbor Electronic Publishing. 2013.

[Mueller2008] Ralf Mueller, Miia Martinsuo and Tomas Blomquist. Project Portfolio Control and Portfolio Management Performance in Different Contexts. *Project Management Journal*. Volume 39, Issue 2, 2008.

[Nielsen1993] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann. 1993.

[Nielsen2012] Lene Nielsen. *Personas - User Focused Design*. Human-Computer Interaction Series. Springer. 2012.

[Norman1986] Donald Norman. Cognitive Engineering. *User Centered System Design: New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum Associates Publishers. 1986.

[Norman2013] Donald Norman. The Design of Everyday Things (Revised and Expanded Edition). Basic Books. 2013.

[Paulhus1991] Delroy L. Paulhus. Measurement and Control of Response Bias. Measures of social psychological attitudes (Volume 1): Measures of personality and social psychological attitudes. Academic Press. 1991.

[Pohl1997] Klaus Pohl and Peter Haumer. Modelling contextual information about scenarios. Proceedings of the 3rd International Workshop on Requirements Engineering: Foundation for Software Quality (RESFQ). 1997.

[Pohl2010] Klaus Pohl. Requirements engineering: fundamentals, principles, and techniques. Springer Berlin. 2010.

[Pressman2009] Roger S. Pressman. Software Engineering: A Practitioner's Approach (Seventh Edition). McGraw Hill. 2009.

[Reid1997] Francis Reid. *Designing for the Theatre*. Taylor & Francis. 1997.

[Ries2011] Eric Ries. The Lean Startup: How today's entrepreneurs use continuous innovation to create radically successful businesses. Crown Pub. 2011.

[Rettig1994] Marc Rettig. Prototyping for Tiny Fingers. *Communications of the ACM*. April, 1994.

[Rice2007] Mark Rice, Alan Newell and Margaret E. Morgan: Forum theatre as a requirements gathering methodology in the design of a home

telecommunication system for older adults. *Behaviour & Information Technology.* Volume 26, No. 4, July-August, 2007.

[Rolland1998] Colette Rolland, Camille Ben Achour, C Cauvet, Jolita Ralyté, Alistair G. Sutcliffe, Neil A.M. Maiden, Matthias Jarke and Peter Haumer. A Proposal for a Scenario Classification Framework. *Requirements Engineering*. Volume 3, Issue 1, 1998.

[Rosson2001] Mary Beth Rosson and John M. Carroll. Usability Engineering: Scenario-Based Development of Human-Computer Interaction. Morgan Kaufmann. 2001.

[Rosson2004] Mary Beth Rosson, John M. Carroll and Con Rodi. Teaching computer scientists to make use. *Putting Scenarios Into Practice: The State of the Art in Scenarios and Use Cases*. 2004.

[Rosson2012] Mary Beth Rosson, John M. Carroll. Scenario Based Design. Human Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications (Third Edition). CRC Press. 2012.

[Schuler1993] Douglas Schuler and Aki Namioka, eds. *Participatory design: Principles and practices.* CRC Press. 1993.

[Sutcliffe2002] Alistair Sutcliffe. *User-Centred Requirements Engineering*. Springer London. 2002.

[Sutcliffe1997] Alistair Sutcliffe. A technique combination approach to requirements engineering. *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*. 1997.

[Vogler2007] Christopher Vogler. The Writer's Journey: Mythic Structure for Writers (Third Edition). Michael Wiese Productions. 2007.

[Watts1996] Nigel Watts. *Writing a novel.* NTC Publishing Group. 1996.

[Weidenhaupt1998] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke and Peter Haumer. Scenarios in system development: current practice. *IEEE Software.* Issue April, 1998.

[Xu2012] Han Xu, Oliver Creighton, Naoufel Boulila and Bernd Bruegge. From Pixels to Bytes: Evolutionary Scenario Based Design with Video. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE).* 2012.

[Xu2013] Han Xu, Oliver Creighton, Naoufel Boulila and Bernd Bruegge. User Model and System Model: the Yin and Yang in User-Centered Software Development. *Proceedings of the ACM International Symposium on New ideas, New paradigms, and Reflections on Programming and Software (SPLASH/Onward).* 2013.

[Xu2015] Han Xu, Stephan Krusche and Bernd Bruegge. Using Software Theater for the Demonstration of Innovative Ubiquitous Applications. Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). 2015.