Software Theater—Teaching Demo-Oriented Prototyping

STEPHAN KRUSCHE, DORA DZVONYAR, HAN XU, and BERND BRUEGGE,
Technische Universität München

Modern capstone courses use agile methods to deliver and demonstrate software early in the project. However, a simple demonstration of functional and static aspects does not provide real-world software usage context, although this is integral to understand software requirements. Software engineering involves capabilities such as creativity, imagination, and interaction, which are typically not emphasized in software engineering courses. A more engaging, dynamic way of presenting software prototypes is needed to demonstrate the context in which the software is used. We combine agile methods, scenario-based design, and theatrical aspects into software theater, an approach to present visionary scenarios using techniques borrowed from theater and film, including props and humor.

We describe the software theater workflow, provide examples, and explain patterns to demonstrate its potential. We illustrate two large case studies in which we teach students with varying levels of experience to apply software theater: a capstone course involving industrial customers with 100 students and an interactive lecture-based course with 400 students. We empirically evaluated the use of software theater in both courses. Our evaluations show that students can understand and apply software theater within one semester and that this technique increases their motivation to prepare demonstrations even early in the project. Software theater is more creative, memorable, dynamic, and engaging than normal demonstration techniques and brings fun into education.

CCS Concepts: • **Social and professional topics** → **Software engineering education**; *Information technology education*; • **Software and its engineering** → **Agile software development**; **Software prototyping**; *Rapid application development*; *Object oriented development*;

Additional Key Words and Phrases: Agile methods, visionary scenarios, scenario-based design, collaborative learning

ACM Reference format:
Stephan Krusche, Dora Dzvonyar, Han Xu, and Bernd Bruegge. 2018. Software Theater—Teaching Demo-Oriented Prototyping. *ACM Trans. Comput. Educ.* 18, 2, Article 10 (July 2018), 30 pages.
https://doi.org/10.1145/3145454

1 INTRODUCTION

Software engineers have to cope with uncertainties and constantly changing requirements (Lehman and Belady 1985). Many educators teach software engineering in a setting close to the real world with industrial customers, which enables students to experience such challenges and prepares them for their later career in industry (Tomayko 1987; Shaw et al. 1991; Bruegge et al.

Authors' addresses: S. Krusche (corresponding author), D. Dzvonyar, H. Xu, and B. Bruegge, Chair of Applied Software Engineering, Department of Computer Science, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany; emails: {krusche, dzvonyar, xuh, bruegge}@in.tum.de.

ACM 1946-6226/2018/07-ART10 $15.00
https://doi.org/10.1145/3145454

2015). Modern capstone courses use agile methods such as Scrum (Schwaber and Beedle 2002) to address uncertainty through an incremental process: software is delivered and demonstrated early in the project in the form of potential product increments or prototypes.

However, a simple demonstration of the functional and static aspects of a prototype fails to provide enough context of real-world software usage, although this is integral to fully understand the software requirements (Shaw et al. 2006). Clarifying requirements is particularly important in exploratory projects with emerging technologies where the real requirements are often unclear until they are met (Carroll 2000). Software engineering involves experimental knowledge work (Basili 1996) such as creativity, imagination, and interaction, but these are typically not emphasized in education. A more engaging, dynamic way of presenting software prototypes is needed to demonstrate the context in which the software is used.

To address this problem, we combine agile methods, scenario-based design (Carroll 1995), and theatrical concepts into a technique called **software theater** (Xu et al. 2015). Software theater is a way to present and evaluate scenarios with a demo of the software under development using techniques borrowed from theater and film, including props[1] and humor. The creation of the demo is based on the Tornado model (Bruegge et al. 2012): developers demonstrate and evaluate requirements, user experience, and the design of the system even if the software is not yet fully realized. Model-based demonstration and the mock object pattern (Mackinnon et al. 2000) allow the simulation of subsystems and objects that collaborate in the demo, but are not yet implemented.

The use of scenarios enables developers to reflect and reason about the requirements before they are realized, by focusing on concrete usage situations in the problem domain and by inspecting them from different perspectives (Carroll 2000). With regular demonstrations, developers can iteratively validate their development progress and technical decisions with the customer and avoid having to change their software at a later stage, which would require higher effort (Hazzan 2002; Shaw et al. 2006).

To examine the benefits of software theater, we investigated the following two hypotheses in two university courses, a capstone course and a lecture-based course:

- **H1 Increased motivation:** software theater increases the motivation of students to prepare a demonstration early in the project.
- **H2 Higher creativity:** demonstrations using software theater are more creative than normal[2] demonstrations (without software theater).

The remainder of this article is structured as follows: Section 2 describes innovation management, interaction design, informal modeling, prototyping, scenario-based design, and the Tornado model as the foundations of software theater. Section 3 illustrates the software theater workflow including all steps from the customer's visionary scenario to the developers' actual demonstration. It introduces the idea of model-based demonstration using the mock object pattern. Section 4 shows an example including all the artifacts created during the workflow. Section 5 presents three recurring patterns: students use a narrator to tell the story behind the system, they use software theater to explain complex technical details of a system, and they make an abstract concept more relatable using a metaphor from a different domain.

[1]The term "prop" is taken from the theater world and describes a theatrical property, an object used on stage or on screen by actors during a performance. It is anything movable or portable on a stage, distinct from the actors, scenery, costumes, and electrical equipment and can be seen as proxy for non-digital objects.

[2]In a normal or traditional demonstration (without software theater), a developer goes through the user interface of an application explaining the functionality of the different user interface elements and the implemented logic (e.g., client server communication) behind it without providing real-world usage context.

We illustrate two case studies in which we taught students with varying levels of experience how to apply software theater. (1) Section 6 describes a large capstone course involving industrial customers with up to 100 students and 10–12 teams per semester (Bruegge et al. 2015). In this course, 60 teams have performed software theater demos in their intermediate and final presentations since 2011. (2) Section 7 presents an interactive lecture-based course about software project management with up to 300 students (Krusche et al. 2017), in which 40 teams have performed software theater demos since 2015. We describe how we teach software theater in both course formats to enable other educators to adopt it in their own courses, and show the results of an empirical evaluation on the use of software theater in both case studies. Section 8 discusses these results with respect to the two hypotheses and provides best practices for instructors who consider to adopt the approach. Section 9 describes related work. Section 10 concludes the article.

2 BACKGROUND

In this section, we describe innovation management, interaction design, informal modeling, prototyping, scenario-based design, and the Tornado model as the foundation of software theater.

2.1 Innovation Management

Software development deals with the improvement of changing and uncertain environments and can be considered as an enabler to innovation management. Innovation is considered as "complex, uncertain, and subject to changes of many sorts" (Kline and Rosenberg 1986). Software projects deal with innovation because the user[3] needs, the technology, and the project environment can change due to external factors. Iterative and incremental approaches allow to start with a manageable set of requirements and to keep improving through validation and refinement cycles.

Validation is a crucial step in the iterative development of software systems; it ensures that innovation goes in the right direction and addresses the correct user needs, integrates the right technology, and provides the best project environment. It reviews whether the user understands the system as it is designed, or vice versa, whether the system is designed as expected by the users (Nielsen 1994). During validation, developers/designers and users need to communicate with the appropriate medium (text, picture or video, story-telling) and pattern (scenario based or not; from the system's viewpoint or from the user's).

2.2 Interaction Design

This communication is necessary so that designers understand how users interact with software systems. Sharp defines interaction design as "designing interactive products to support people in their everyday and working lives" (Sharp 2003). Norman introduced the terms `Design model` and `User model` to describe the mental understanding of designers and users and their interaction with the system (Norman and Draper 1986).

The `Design model` describes the designer's mental model of how the user will use the system through its user interface. A mental model is based on tacit (as opposed to explicit) knowledge and is difficult to transfer to other persons by means of writing or verbalization. Since the user only interacts with the user interface of the system, the design model focuses on the interface and interaction between user and system.

The `User model` presents the user's mental model of how the system will work from the usage perspective. While a user can have existing knowledge about the problem domain, this model is also formed by interacting with the system and by reading the documentation (e.g., user manuals).

[3]A user is an external stakeholder using the system. Different roles and types of users can be distinguished.

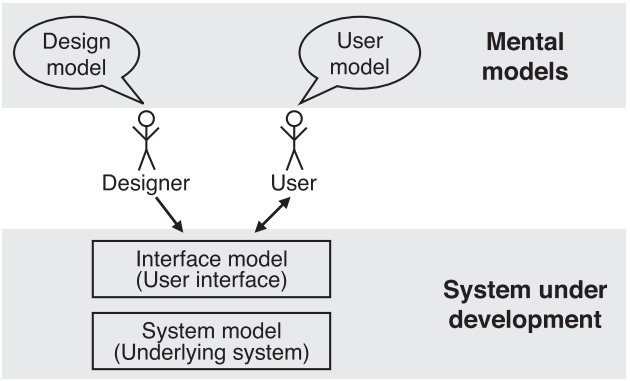

Fig. 1. The relationship between mental models and system under development (adapted from Norman and Draper (1986)).

Figure 1 shows these two mental models and their relation with the `Interface model` and the underlying `System model`.

The `Interface model` helps to emphasize the system components that are relevant to the end user and is separated from the system model. It hides the underlying details that are invisible to the user and is about the user-visible aspects of the system.

The `System model` provides an abstraction of the implementation and is used as a convention for communication among the developers. System modeling allows to focus only on the interesting aspects of a complex system and ignore irrelevant details (Bruegge and Dutoit 2009). System models are typically documented and conveyed in the form of Unified Modeling Language (UML) diagrams.

Design model and user model are mental models, i.e., "conceptual models formed through experience, training, and instruction" (Norman 2013). In a well-usable software system, the user model is consistent with the design model and is addressed by the interface model and the system model. User involvement helps to achieve usable systems in the software design process. It aims to evaluate the design and obtain feedback from users about the consistency of the design model with the user model.

2.3 Informal Modeling

Models are used for different purposes such as specification and documentation, but also facilitate the communication between developers. Formal models, such as UML diagrams, are used to specify and document the software system completely, consistently, and correctly. However, the creation of such formal models is usually time-consuming. Developers resist to change the model if they have invested a lot of time to create it. Small changes can lead to huge effort if all formal models have to be updated to ensure consistency. If changes occur, the time spent to create the formal model might have been worthless. Formal models are difficult for users without a technical background to understand (Pressman 2009). They cannot be used to validate the user model and the design model.

Informal models can be created faster and changed easier because they do not focus on completeness, consistency, correctness, and formality. They might not follow formal notations and include inconsistencies or vagueness for parts of the design that have not yet been realized. They are adapted incrementally and iteratively and are often used in agile methodologies to reduce the effort for comprehensive documentation as they focus more on communication (Beck et al. 2001).

They typically describe the look-and-feel and the interaction with the system from the user's point of view (Bruegge et al. 2012). Examples of informal models include sketches, paper prototypes, low-fidelity user interfaces, storyboards, and text-based and video-based scenarios (Xu et al. 2013).

2.4 Prototyping

Prototypes are examples of informal models and can be used to validate if the requirements are understood correctly. There are different prototyping techniques and tools to "involve an early practical demonstration of relevant parts of the desired software" (Floyd 1984). Demonstrating the design to stakeholders and expecting feedback from them provides validation at different levels: requirements, user experience design, and technical architecture design. The demonstration can be conducted based on prototypes of different fidelity levels depending on which is appropriate in the given situation. Ideally, the design should be demonstrated and evaluated when there is a change that may cause significant consequences.

Compared to fully implemented systems, prototypes allow to evaluate design ideas more quickly and at lower costs. This is achieved by defining appropriate focus and choosing the right form of prototype for the given situation (Arnowitz et al. 2010). As prototypes alone do not provide enough context of the usage, scenarios can be used as a complement (Weidenhaupt et al. 1998). Scenarios are concrete descriptions of the system usage and are helpful in making sound design decisions by focusing on both the problem space and the solution space (Jarke and Pohl 1993; Jarke et al. 2010).

2.5 Scenario-based Design

Another example for informal models are scenarios, which are used as examples for illustrating common use cases and requirements of the system. Their focus is on understandability and communication. Different types of scenarios exist. Visionary scenarios describe a future system: they are used by developers to refine their ideas and as a communication medium to elicit requirements from users. They can be viewed as an inexpensive prototype. As-is scenarios describe an existing system or a current situation. They can be validated for correctness and accuracy with the users of the corresponding system/approach. Demo scenarios focus on the demonstration of a system/approach: they describe a future system that is currently implemented and needs to be validated.

Scenario-based design is a development approach that uses scenarios to design the future system. It focuses on the users of the system and their interaction with the system. Rolland and his colleagues state that "people react to 'real things' and ... this helps in clarifying requirements" (Rolland et al. 1998). Scenarios are intuitive and suitable for communication and validation. The story-like description with context information makes it easier for stakeholders to understand abstract concepts in the system design. Scenarios are cheap to create and enable quick iterations. In a changing environment, the design of the system often takes several revisions to reach a "stable" state. Scenarios are also open-ended and stimulate the user's imagination. They enable the users to come up with more specific requirements and help "the analysts to consider contingencies they might otherwise overlook" (Carroll 2000). There are different ways to use prototypes with scenarios depending on who actually demonstrates the scenario. It can be user-performed, where users are provided with scenarios as description of tasks and are told how to use the prototype to perform the task (Pohl 2010), or developer-performed, where scenarios provide a context in which the prototype demonstrates how to achieve specific tasks (Sutcliffe 1997). Weidenhaupt and his colleagues report that 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 (Weidenhaupt et al. 1998). This is particularly important in innovative applications based on emerging technologies (such as wearable computing, Internet

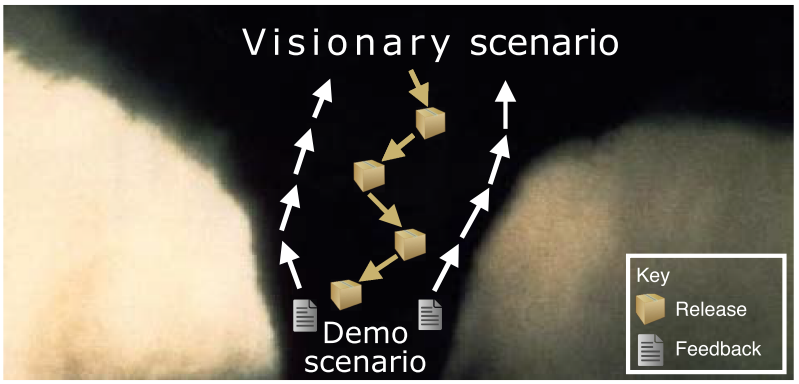

Fig. 2. Tornado model: wide in analysis, narrow in implementation with multiple releases funneling down and feedback as updrafts (Bruegge et al. 2015).

of Things, augmented reality, virtual reality) because the desired applications have not been seen or even imagined by users before. The IKIWISI (I'll know it when I see it) principle (Boehm 2000; Cao and Ramesh 2008) must be considered.

2.6 Tornado Model

The Tornado model, shown in Figure 2, describes how visionary scenarios can be transformed into demo scenarios in order to show an executable system that represents the vision under evaluation (Bruegge et al. 2012). It is a demo-oriented development process aiming to deliver "touchpoints," a metaphor for creating executable prototypes that evaluate design ideas and obtain feedback from the stakeholder. Informal models are good at closing the communication gap between developers and users because they are easier to create, understand, and revise. The Tornado model highlights the importance of informal models in addition to formal models. This is also expressed with the idea of yin-yang balance in software development (Xu et al. 2013).

Visionary scenarios represent design ideas of systems, and are used for requirements brainstorming. They require several rounds of iterations to reach a stable version. The main task at this stage is exploring the problem space, so low-fidelity prototypes are sufficient. Demo scenarios are refinements of visionary scenarios for reviews and presentations. They provide a demonstration how the core of the problem is realized when using the system and can be played out in a demo. Demo scenarios are based on a (partially) working system and take advantage of mockups for cost-efficiency reasons.

The Tornado model describes an evolutionary scenario-based design process. The initial version of the design is depicted using low-fidelity prototypes, used in the early stages to obtain user feedback about the user interaction design. This enables the user to explore possible design alternatives and reformulate the initial requirements. In the middle of the project, when more stable design alternatives have been chosen, interactive prototypes (e.g., created with Balsamiq[4]) are used for a more tangible and reliable evaluation. At the end of the project, the finally adopted design is implemented and delivered using the tornado metaphor: A tornado is wide in the clouds (vision), but only a part of it funnels down and hits the ground at its touchpoint (demo). The touchpoint is where an executable demo system is created and presented. Feedback, obtained through the demo, influences the visionary scenario as updrafts of the tornado.

[4]Balsamiq is a digital low-fidelity mockup tool with support for executable prototypes: http://balsamiq.com.

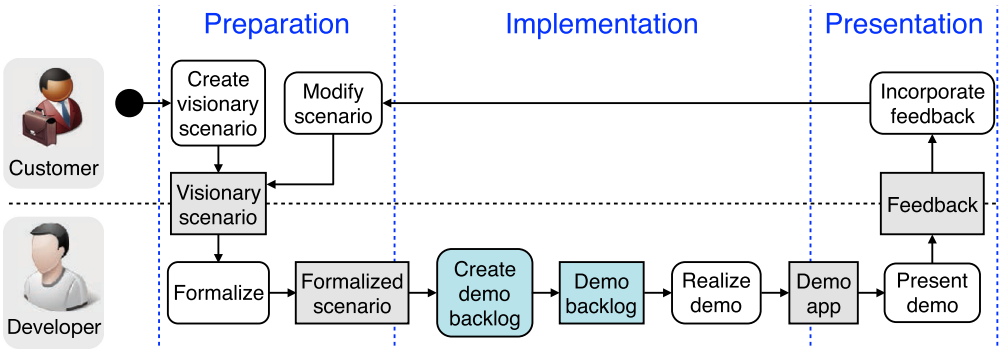

Fig. 3. Main activities and artifacts in the software theater workflow divided into three phases represented with swim lanes (UML activity diagram). The blue elements are explained in further detail in Figure 4.

3 SOFTWARE THEATER WORKFLOW

Similarly to performing a prototype based on predefined scenarios, software theater is performed based on a theater screenplay. The screenplay describes the event flow of the demo, the cast (participating actors), and the props required for the demo. The purpose of software theater is to demonstrate how end users would benefit from the new product in a real-world context. Figure 3 shows the full workflow for creating a demo using software theater. The workflow is divided into three phases represented with UML swim lanes: preparation, implementation, and presentation.

3.1 Preparation

In the beginning of the project, the customer creates the problem statement, which includes `Visionary scenarios`. The development team prioritizes these visionary scenarios with the customer. Each scenario is formalized using a template with six components: scenario name, participating actors, flow of events, entry conditions, exit conditions, and quality requirements. A `Formalized scenario` is the basis for the demonstration. It describes the same content as the visionary scenario, but in a structured way (Bruegge and Dutoit 2009): "Formalization helps to identify areas of ambiguity as well as inconsistencies and omissions in a requirements specification."

3.2 Implementation

After the preparation, the developers start to implement the demonstration. This phase consists of the activities `Create demo backlog`, which involves multiple sub-activities and `Realize demo`. Figure 4 shows the detailed actions and artifacts for the activity `Create demo backlog`.

The team writes the `Screenplay` (called demo script) based on the event flow as well as the participating actors of the formalized scenario, and identifies the props and stage directions needed for the scene. The developers select the subsystems and services that are required to realize the demo based on the system architecture. The focus of the demo is on subsystems that require technical validation (e.g., performance-critical or features related to user experience).[5] Less critical subsystems can be mocked to save development time for the demo.

The team identifies participating methods and participating objects in the selected subsystems by inspecting the flow of events in the formalized scenario. It uses textual analysis (e.g., Abbott's technique) to identify nouns as candidates for classes (participating objects) and verbs as

[5]The selected subsystems are highlighted in the subsystem decomposition (UML component diagram).

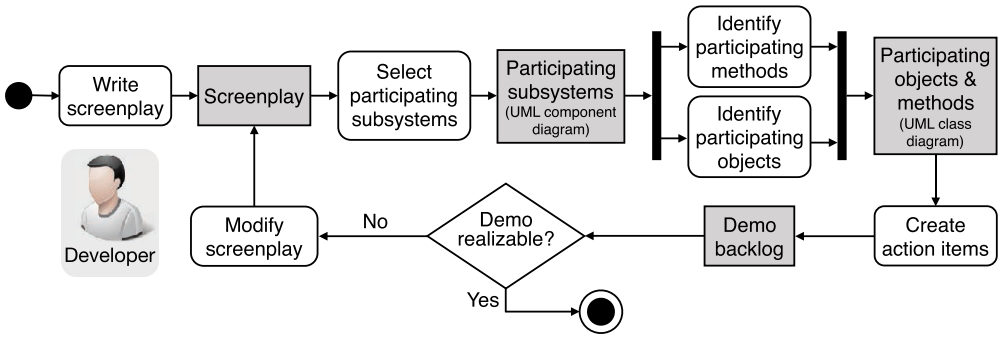

Fig. 4. Detailed actions and artifacts for the `Create demo backlog` activity (UML activity diagram).

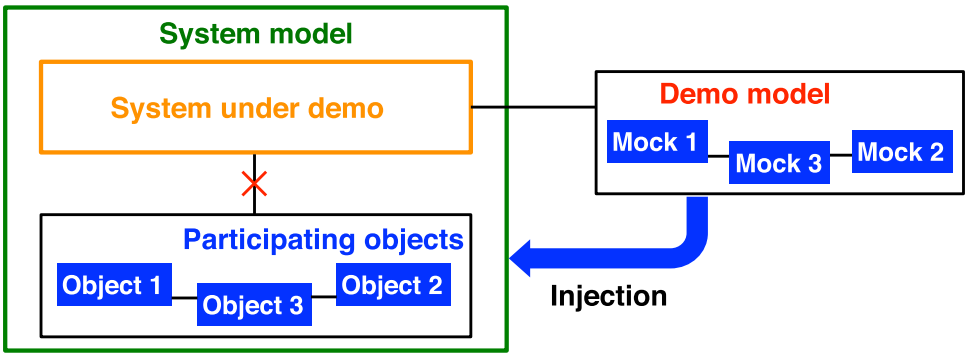

Fig. 5. Model-based demonstration using the mock object pattern: the demo model with mocks replaces participating objects that are not implemented yet.

candidates for operations (participating methods) (Abbott 1983). This identification can be performed in parallel and includes the creation of a UML class diagram that includes these objects and methods. Mocked subsystems, objects, and methods are highlighted in the class diagram so that all developers are aware of the decision.

Developers apply the mock object pattern (Mackinnon et al. 2000) in which real collaborators (i.e., participating objects) are replaced by mock collaborators as shown in Figure 5. Based on the concept of model-based testing (Apfelbaum and Doyle 1997), we call this technique **model-based demonstration**. It allows to focus on the implementation of the identified objects and methods (`System under demo`) and to mock other parts of the system that are not relevant. The dependency injection pattern (Martin 1996) allows to switch between mock and real implementations during development.

The resulting `Demo backlog` contains all the action items for the implementation of participating objects and methods and represents the task model and management aspects behind the workflow. The demo backlog also includes the generation of collaborating mock objects and the preparation of props. After the creation of the demo backlog, the developers assign the action items and estimate if they are able to realize all action items before the demonstration.

If they answer this question with "Yes," they start with realizing the demo. Otherwise, they need to modify the screenplay and leave out certain aspects in the demonstration or mock additional subsystems, objects, and methods. The realization of the demo app can follow a continuous integration and continuous delivery process. During the realization, the developers tick off

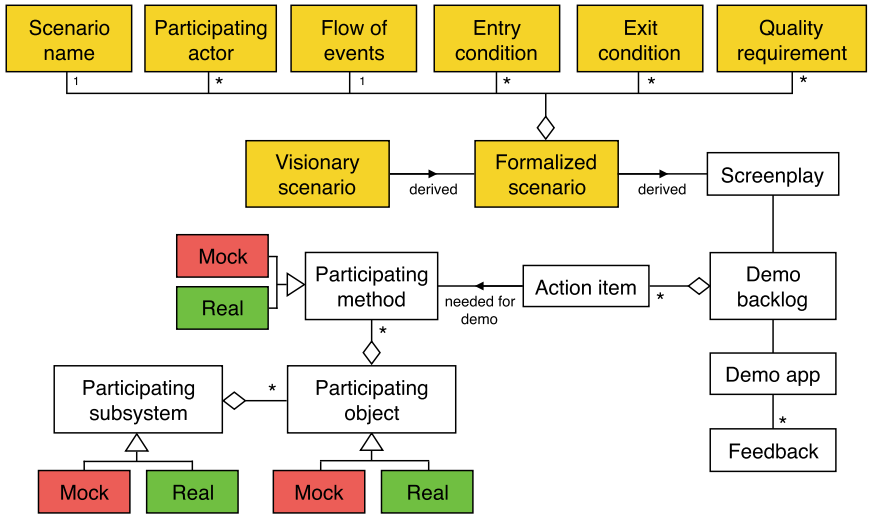

Fig. 6. Software theater's main concepts and relationships in an analysis object model (UML class diagram).

finished action items or further refine the demo backlog. After the realization and the delivery, the workflow proceeds to the presentation phase.

3.3 Presentation

During the presentation, team members assume the roles of the actors of the screenplay. After the demo, the team collects feedback and incorporates it by modifying the visionary scenarios. Typical feedback includes user interface issues and conceptual aspects of the system; it usually leads to changes in the prioritization of the visionary scenarios. Sometimes, it even requires a change of the used technologies.

Figure 6 shows all artifacts of the software theater workflow as objects in an analysis object model. This illustrates how these artifacts are related to each other. For instance, the action items are connected to participating methods that need to be realized for the demo. The action items therefore connect the task model (demo backlog) with the object model of the developed system that is visualized in a UML component diagram (participating subsystems) and in a UML class diagram (participating object and participating methods).

4 SOFTWARE THEATER EXAMPLE

To exemplify this workflow, we show an example taken from our capstone course (cf. Section 6). It includes all artifacts of the software theater workflow for one concrete project of the course.

The project was done in collaboration with Zeiss Meditec,[6] an industry partner in the field of medical technology. Its goal was to develop the Zeyes app to digitize the process of stock taking and re-ordering of lenses for cataract surgeries at hospitals around the world. One of the visionary scenarios received from the customer was as follows:

> *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.*

[6] Zeiss Meditec is a department of Zeiss, specializing in medical technologies: https://www.zeiss.com/meditec.

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	

Fig. 7. Formalized scenario for the stock taking demo with the flow of events as basis for the screenplay.

> *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 introduce organizational issues and costs, and every minute that she saves in this process can be used to have value added discussions with the customer.*

The team selected parts of this scenario for their demonstration and formalized it. Figure 7 shows the formalized scenario with name, participating actors, flow of events with user and system steps, as well as entry and exit conditions (Bruegge and Dutoit 2009). This concludes the artifacts of the "preparation" phase of the workflow.

The team wrote a corresponding screenplay for their demonstration. It chose to have a narrator who leads the audience through the scene and two employees who perform the stock taking process. During the theater, Christina manually takes the stock and gets frustrated by its inefficient and confusing nature, while Matthias uses the Zeyes application, which enables him to scan the products by barcode, see which products are missing in the inventory, and send a report with the results. Figure 8 shows this story as screenplay. After writing the screenplay, the students continued with the selection of participating subsystems, and the identification of participating objects and methods. They serve as basis for creating the demo backlog. The team did not document these steps, as they only went through them informally. Therefore, we used their system models to create possible representations of the artifacts in this phase of the workflow to complete this example.

Figure 9 shows a simplified version of the subsystem decomposition that focuses on the components included in the demo. The `Smartphone App` runs on the customer representative's smartphone at the customer location (e.g., a hospital). It uses the `Smartphone Camera` to identify items by barcode and an `NFC Reader` (near field communication) to scan items by NFC tag. It communicates with the `Customer Service Hub` to retrieve necessary customer information, to submit reports, and to place orders as a result of stock taking.

(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)

Fig. 8. Excerpt of the screenplay for the stock taking demo with the narrator, Christina, and Matthias as actors. The screenplay starts on the left side and continues on the right side.

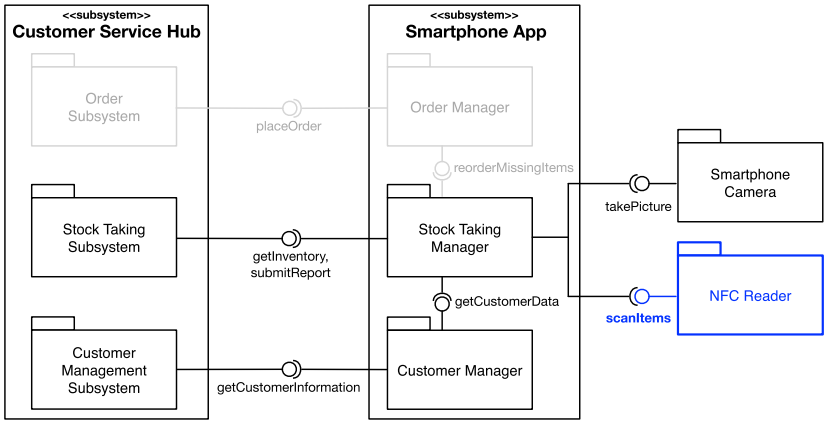

Fig. 9. Selected participating subsystems for the stock taking demo (UML component diagram). Not participating subsystems are shown in grey. The blue subsystem NFC Reader is mocked.

The demo scenario at hand did not include submitting an order, which means that the `Order Manager` and the `Order Subsystem` are not relevant for preparing the demo and are thus greyed out in the UML model in Figure 9. The team knew already that they would mock the `NFC Reader` because they would not have the necessary accessory to read NFC tags on the iPhone. By realizing early which parts of their system are most relevant for the demo, the team could concentrate on implementing the critical functionality. The next step of the workflow involved the identification

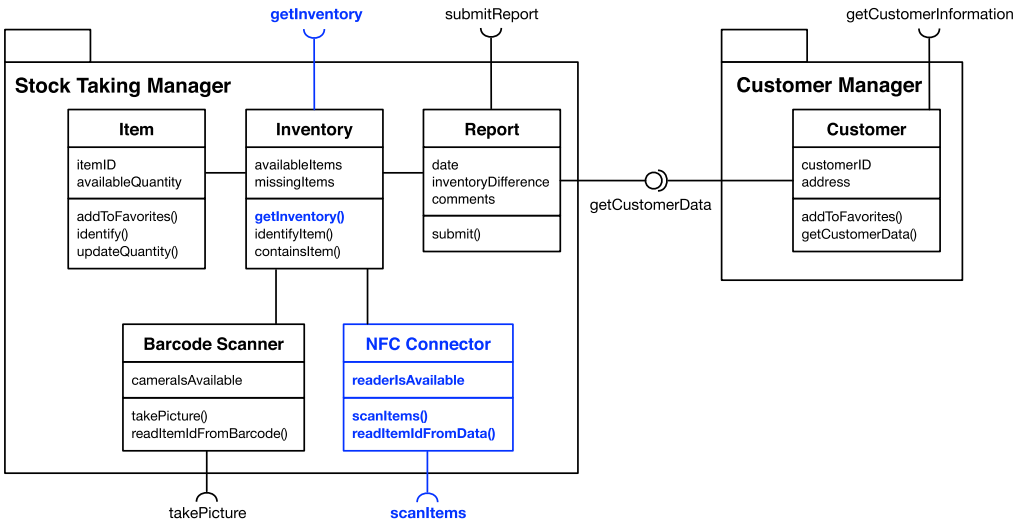

Fig. 10. Identified participating objects and methods for the stock taking demo (UML class diagram). Blue elements are mocked.

of participating objects and methods. Starting from the selected demo-critical subsystems, the developers identified the corresponding objects in these subsystems that they needed to implement for the demo. Figure 10 shows the result of this step, simplified to include only the two components of the Zeyes app that are required for the demo.

The developers decided which objects and methods can be mocked for demonstration purposes. For instance, they might need to mock a component because they do not have the time to fully implement it or because there is a technological constraint which keeps them from implementing it. Another reason could be that they want to reduce the risk of demo failures, e.g., in complex setups with distributed components. The team mocked the component `NFC Reader` because it did not have a reader to connect to the iPhone at the time. It implemented a mock object `NFC Connector`, which always indicates that the reader is available and returns the same, pre-defined data when scanning items.

The developers also mocked the interface `getInventory` provided by the `Stock Taking Subsystem` in the `Customer Service Hub` (cf. Figure 9) because the data of real server responses contained cryptic item names. They changed the `getInventory` method in the `Inventory` object to return more understandable item names for their live demo. The mocked elements are marked in blue in Figure 10.

After identifying the participating methods and objects, the team created the demo backlog including implementation tasks and organizational tasks. These tasks should be small enough so that they can be easily distributed to individual team members. Figure 11 shows an excerpt of the demo backlog for this demonstration, including action items for implementation, for the creation of mocks, and for the preparation of props.

The team performed their demonstration on February 2, 2017. Figures 12 and 13 show scenes of the live demonstration where two sales representatives interacted with the developed software and with each other.[7]

[7] A recording of the demo is available at http://youtu.be/MTayd0kyY6Y (the demo starts at 06:30).

- [ ] 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)
- [ ] …

Fig. 11. Demo backlog with tasks to realize for the stock taking demo.

Fig. 12. Software theater demo: actors Christina and Matthias perform the scenarios for manual and digital stock taking (Zeiss Meditec project).

Fig. 13. Software theater demo: Matthias explains to Christina the advantages of the Zeyes app that enables digital stock taking (Zeiss Meditec project).

5 SOFTWARE THEATER PATTERNS

Based on the patterns idea (Alexander et al. 1977), we identified three software theater patterns that highlight the creativity and flexibility of software theater demonstrations.

5.1 Narrative Pattern

The narrative pattern describes the role of a narrator in the software theater scene. Teams typically use this pattern to leverage its capability for detailed explanation: for instance, the narrator can set the scene and explain how and why the system solves a problem in the particular situation.

An example of the application of the narrative pattern was the Zeiss Industrial Measuring Technology (ZIMT) project. It addressed the problem of monitoring expensive machines shipped to the customer around the world using multiple transport methods such as trucks, container ships, and sometimes even elephants. To know whether the shipping company is liable for a damage of the machine upon arrival, the customer accompanies each machine with a sensor, measuring shocks in all three axes, as well as deviations in temperature and humidity. The team developed a corresponding smartphone application. A Zeiss service mechanic can use it to read the sensor data and get a first clue on where to start inspecting the damaged machine.

The demonstration showed the machine in a closed box and the collection of sensor data was not an instantly visible process. Therefore, the team made the problem more obvious by involving a narrator who explained what was happening from the perspective of the machine. They even used the first person to make the situation more relatable. For instance, the scene started with the machine introducing herself and arguing: “Because of my high value, I have a sensor with me. It

tracks shocks and extreme temperature values. Zeiss uses it to keep an eye on my status." As a result, the actors could play their parts without explaining the rationale behind their actions.[8]

5.2 Explanation Pattern

The explanation pattern shows a technical detail of the system within a software theater scene. Some teams illustrate an algorithm, while others explore the details of a particular technology. An example for this is the Allianz project in the winter semester 2014/15. The application involved the identification of intrusions at home with the help of multiple sensor readings. The team used the blackboard architectural pattern (Buschmann et al. 1996) to detect an intrusion such as a burglary from the sensor readings. They brought in a superhero named AllianzMan symbolizing the intelligence of the system to explain how this pattern works.

AllianzMan coordinated several experts, illustrating the specialized knowledge sources of the blackboard pattern, such as a sensor change expert or an occupant identification expert. Each of these experts was played by a team member. In their demo, an intruder accessed the building while the home owner was away, leading to unusual sensor readings such as opening doors, lifting valuable objects, and raising noise levels. The experts took turns passing the values back and forth, analyzing and discussing them to combine their knowledge, and getting closer and closer to a solution under uncertain conditions. When they decided that this was likely an intrusion into the home, they alerted the home owner. Figure 14 shows a scene of the demo.[9]

5.3 Metaphor Pattern

Another software theater pattern is the use of a metaphor (e.g., a concept of a different domain) to explain the purpose of the system in a relatable way. This pattern is typically used when the system addresses a problem that only occurs in a certain field or for a particular kind of user, or if the concept behind the system is abstract and not easily understandable.

The BSH project in the winter semester 2016/17 used this pattern in their demonstration of their system called "Scentdipity." The app was built to help people design perfumes from home, introducing personalization and new opportunities for creativity. To illustrate the concept of creating a perfume out of solvents and scents coming from different olfactive families, the team chose to demonstrate their system with a blender mixing smoothies.

They adapted their application to show only scents based on fruits and explained why it was important that the system hides bad combinations of ingredients, or helps to select the right amount of solvent (symbolized by a carton of milk) for the perfume. This made it easier for end users to create a favorable result and to feel like the product is their own creation at the same time. The actress also showed how the Scentdipity machine (illustrated by the blender) could mix small amounts of the desired perfume for testing purposes. Figure 15 shows an impression of the demo.[10]

6 CASE STUDY 1: CAPSTONE COURSE

We teach software theater in our regular capstone course "iPraktikum," which takes place twice per year at the Technical University of Munich. In this course, 80–100 students develop systems involving a mobile component for a real customer from industry. The course is based on a multi-customer organization (Bruegge et al. 2015) with 10–12 projects running at the same time, and a continuous process model, Rugby (Krusche 2016). Rugby extends Scrum with continuous delivery workflows and was adapted for university courses to account for part-time developers.

[8] A recording of the demo is available at http://youtu.be/enpEO6bGaZQ (the demo starts at 01:55).
[9] A recording of the demo is available at https://youtu.be/efHEQQaVp6U (the demo starts at 04:20).
[10] A recording of the demo is available at http://youtu.be/85goFfxUkuk (the demo starts at 03:50).

Fig. 14. Illustrating the knowledge sources of the blackboard pattern with software theater using the explanation pattern (Allianz project).

Fig. 15. Borrowing concepts from other domains in software theater using the metaphor pattern (BSH project).

6.1 Course Description

Each project team consists of a project leader, who is responsible for the project outcome as well as grading the participants. The project leader works together with a team coach; this is a student who has previously participated in the course as developer, assumes the role of a scrum master, and takes care of day-to-day problems and agile processes. Each team consists of six to eight developers, some of which take an additional functional role, involving tasks or workflows such as modeling, release management, or merge management that are important across all projects. Functional teams meet regularly to discuss their respective topic and bring the knowledge back into their project team. In course-wide lectures, we cover topics concerning all participants, e.g., software theater.

We describe the structure and workflows of the course in more detail in Bruegge et al. (2015). Setting and structure of the course are very close to industry and address the majority of the gaps identified by Nurkkala and Brandle (Nurkkala and Brandle 2011). We have held 12 iterations of the course, including over 110 distinct projects with different customers from industry,[11] since its introduction in 2008.

We studied the use of software theater during a capstone course running from October 2016 to February 2017. In this instance, 91 students (80 developers and 11 coaches) participated in the course, and were distributed into 11 teams, each between 8 and 9 participants. We started the course with a kickoff meeting on October 20, 2016, in which the industry partners presented their problem to all participants. We then allocated the students to teams based on their project priorities and prior experience, taking into account the importance of balanced teams for learning (Bruegge et al. 2015). An excerpt of the organizational chart with five teams of the course is shown in Figure 16. In the following 2–3 weeks, each team went through a Sprint 0, an iteration where the focus is not on development, but on understanding and analyzing the problem (Bruegge et al. 2012; Dzvonyar et al. 2014).

Eight weeks after the beginning of the course, all teams presented the results of their requirements analysis and system design activities in the design review, an intermediate milestone. While we emphasized the importance of showing the architecture and technical details of the system, we encouraged the participants to also incorporate a demonstration using software theater in their presentation. We believe that the demonstration of an executable prototype early in the project leads to a better understanding of the project status and more realistic feedback from clients and

[11] Descriptions and videos for all projects: https://www1.in.tum.de/lehrstuhl_1/projects/all-projects#iOS.

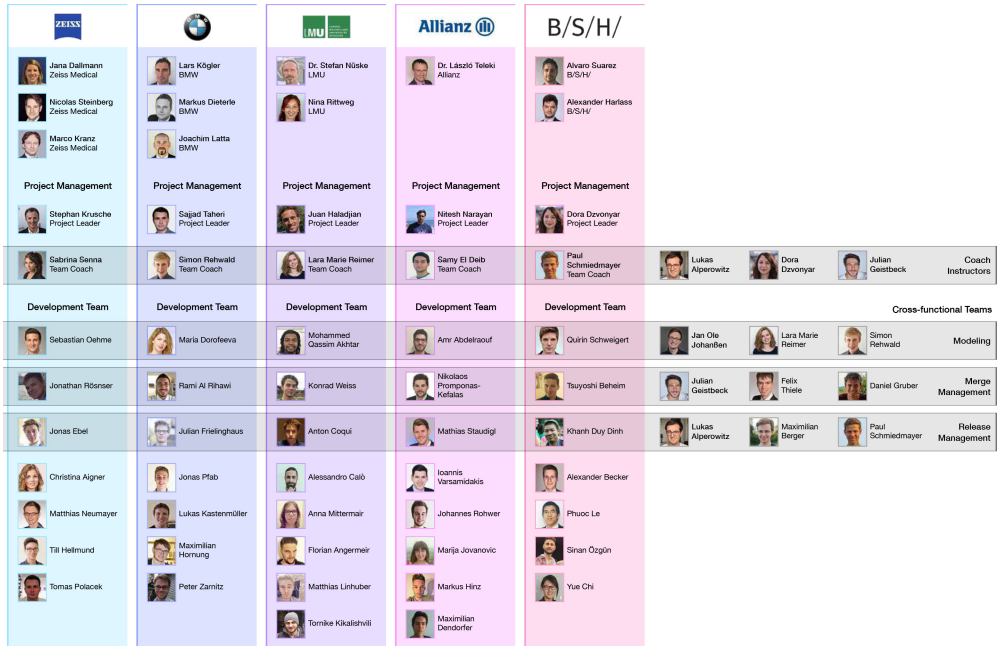

Fig. 16. Excerpt of the organizational chart of the capstone course in the winter semester 2016-17.

other stakeholders. The teams can use this feedback to refine the requirements for the following iterations (Bruegge et al. 2015).

To teach the participants the necessary knowledge about software theater, we held a course-wide lecture in which we introduced the software theater workflow, showed them examples, and provided recommendations and best practices on how to prepare their demo. This lecture took place 3 weeks before the design review so that the students had enough time to apply the workflow and to iterate over their demo and presentation. The teams rehearsed the theater play multiple times and iteratively adapted the interaction between the users and the developed demo app. We used Prototyper (Alperowitz 2017), which allows the repeated delivery of executable prototypes to the target environment, in our case the demo apps to the theater stage. We provided feedback on their presentation in a dry run one week before the design review.

After the design review, the teams continued development for another 7 weeks and presented the final outcome of their project in the client acceptance test on February 2, 2017. In these presentations, we asked the teams to focus on a meaningful demonstration of their system that is understandable to a broad audience. We also encouraged them to perform regular demonstrations to their customers using software theater in-between the course-wide events. The presentation recordings of the design review and client acceptance test are available on our web site.[12]

6.2 Evaluation Design

We closely observed the teams during our capstone course in the winter semester 2016/17 to analyze the impact of software theater. We measured the following variables at our main events, the design review, and client acceptance test:

[12] http://www1.in.tum.de/ios1617.

Table 1. Overview of Software Theater Usage for Each Team in the Design Review and the Client Acceptance Test

	Team name	Allianz	BMW	BSH	iHaus	LMU	McKin	Quart	T-Sys	ZIMT	Zeyes	ZF
	Number of developers	8	7	7	8	8	7	7	7	7	7	7
Design review	Application	yes	yes	no	yes	yes	yes	yes	no	yes	yes	yes
	Demo length [min]	2.6	3.6	3.6	2.1	2.4	2.6	2.3	0	3.4	2.8	1.1
	Demo participants	2	3	1	4	3	3	4	0	2	2	5
	Integration	no	yes	no	yes	no	no	no	no	no	no	no
Client acceptance test	Application	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
	Demo length [min]	4.4	2.6	5.4	2.5	3.3	3.5	3.7	9.1	6.2	3.5	3.2
	Demo participants	3	3	3	4	6	3	4	4	4	2	3
	Integration	no	no	no	yes	no	no	no	yes	yes	no	yes

(1) **Application:** whether teams used software theater for their demonstration
(2) **Demo length:** the length of the demonstration in minutes
(3) **Demo participants:** the number of participants in the demonstration
(4) **Artifacts:** which artifacts of the software theater workflow the team produced and documented
(5) **Integration:** the integration of the demo into the presentation

We regularly verified our observations in the development tools the teams used for communication and documentation. For instance, we checked whether the teams documented software theater artifacts in their team wiki. In addition to these observations, we evaluated the usage of software theater and the personal opinions of the students in an online questionnaire. After the client acceptance test, we sent out a survey to 80 developers and 11 team coaches who participated in the course. The survey included questions about the demonstrations in both major presentations. In particular, we had the following areas of interest:

(6) **Role:** the respondent's role at each event (demo participant, presenter, or none)
(7) **Preparation time:** the amount of time the respondent spent on the preparation of each demonstration
(8) **Mocks:** whether parts of the system were mocked and, if yes, why they were mocked
(9) **Software theater value:** a comparison of a demo using software theater to a normal demonstration
(10) **Software theater characteristics:** opinion on software theater as an approach to realistic demonstrations

We combined the observation and the questionnaire to get both a first-hand account of the students' experience using software theater, and a more reliable and complete overview of the usage of the approach in the case study.

6.3 Evaluation Results

We analyzed the demonstrations and measured the above variables for each of the 11 teams. Table 1 summarizes the results of the evaluation for both events: `Design review` and `Client acceptance test`.

In the design review, the teams dedicated on average 2.4 minutes out of the 10 minutes of presentation time to perform a live demonstration of their early software prototype. Nine out of the

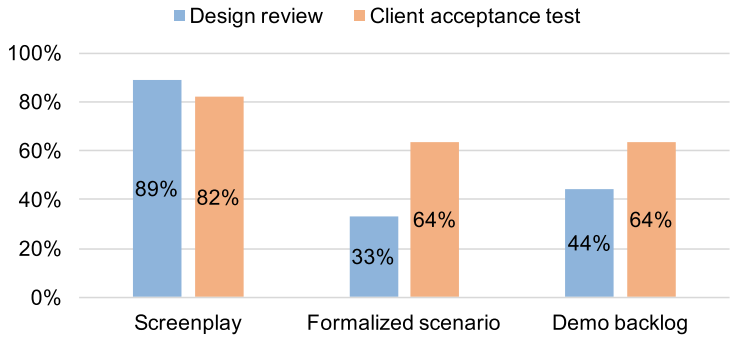

Fig. 17. Percentage of teams who produced software theater artifacts in the capstone course (grouped by event).

eleven project teams used software theater. Team BSH chose to narratively walk through their current state with only one demo participant. Team T-Systems skipped the demonstration because they felt they had not made enough progress with the software due to hardware challenges they needed to overcome in the first weeks of their project. Two of the teams designed their presentation from ground up with the demonstration in mind, integrating it into their story in multiple parts. For instance, team iHaus performed the first part of the demo and then interrupted it to explain a technical detail of their application before carrying on with their software theater. This allowed them to use the practical demonstration of their system to support their arguments and explanations in the presentation.

In the client acceptance test, all teams used software theater, with an average demo length of 4.3 minutes, which accounted for almost half of their presentation time. Team T-Systems, who had not included a demonstration in the design review, now chose to perform their whole presentation with software theater, explaining their requirements and project state through their role play. The number of teams that interweaved demo and presentation doubled in the client acceptance test, with teams using their live demo to explain technical details of their system in depth as described with the explanation pattern in Section 5.2. Team ZF performed their software theater in three parts, with interruptions explaining their algorithms for the detection of dangers while driving (detection and warning of intersections, gaze detection, and alert of distracting audio levels in the car). All teams had at least two demo participants; one team had six (3.5 on average).

Figure 17 shows how many teams produced software theater artifacts throughout the project. The majority of the teams (89%) wrote a screenplay and up to 64% created a formalized scenario as well as a demo backlog.

In the survey, we received 80 responses (70 developers and 10 team coaches, response rate: 88%). We were interested in the amount of preparation that went into the demonstration at each event, including producing the artifacts of the software theater workflow, preparing mock data, practicing the demonstration, preparing props, and iterating over the demo or giving feedback to fellow team members. Figure 18 shows the responses to this question. For the design review, 36% of respondents indicated that they spent 3 hours or less to prepare for the demo, while 24% invested 10 or more hours. The preparation time increased for the client acceptance test: 37% of participants prepared more than 10 hours, while 29% spent less than 3 hours on the demo.

We asked the developers and coaches to compare a demonstration with software theater to a normal software demo using a pre-defined set of adjectives. As illustrated in Figure 19, over 80% of respondents think that a demonstration using software theater is more creative, fun, and memorable than a demo without the technique, and two-thirds think that it is more dynamic

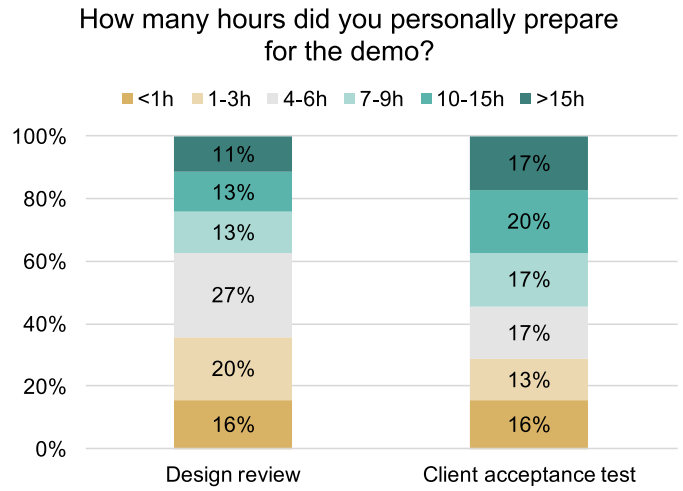

Fig. 18. Survey responses: preparation time for the client acceptance test was higher than for the design review.

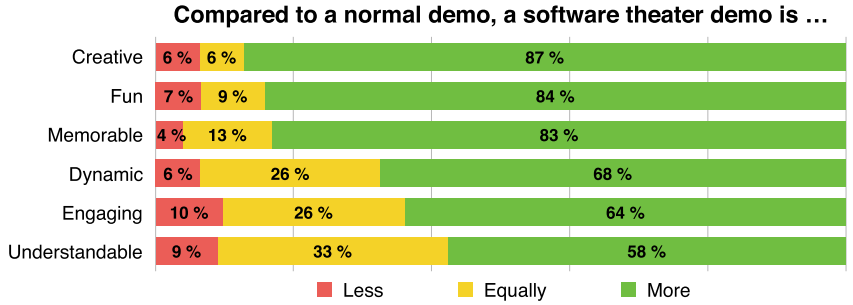

Fig. 19. Survey responses: the majority of students thinks that software theater is more creative, fun, memorable, dynamic, engaging, and understandable compared to a normal demo.

and engaging. Less students, 58%, think that software theater makes a demonstration more understandable.

This is also visible in the responses to the set of 3-point Likert-scale questions shown in Figure 20: while most participants agree with having an improved understanding of *other teams'* systems through software theater, their opinions are divided concerning *their own team.* 34% of respondents agreed that software theater helped them to understand their own project's requirements, with roughly the same amount of neutral responses and disagreeing participants. More participants disagree than agree that software theater helped them to communicate with their client. However, 55% agree that their demonstration using software theater gave them confidence about their system's usefulness, and the majority of respondents will use the approach to create future demonstrations.

71% of the respondents indicated that they mocked parts of their system using the mock object pattern. Among the most frequently stated reasons were the following:

(1) No access to components necessary to realize a certain functionality, e.g., NFC reader, sensors, or an external machine to communicate with.

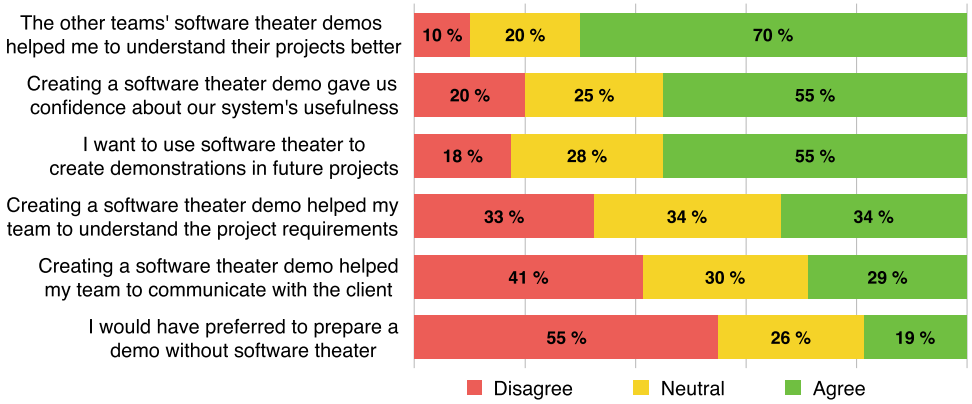

Fig. 20. Survey responses: disagreement (red) and agreement (green) with software theater characteristics on a 3-point Likert scale.

(2) Fallbacks for connectivity issues, e.g., mocking server responses in case the internet connection fails during the demo.
(3) Fallbacks for complex computation, e.g., complex sensor data analysis, which would have taken too long for the demo.
(4) Access to data that is not possible to get during the demo, e.g., historical GPS data, which would otherwise not be present on the demonstration device.

We discuss these findings in Section 8.

6.4 Threats to Validity

One limitation of our evaluation is that we did not have a control group within the same course, because all teams in our capstone course use software theater. Another threat is that we used Likert scales, which may be subject to distortion (Garland 1991). For instance, respondents may have avoided using extreme response categories (central tendency bias) and they may have agreed with statements as presented (acquiescence bias). They also may have tried to portray themselves or our course in a more favorable light (social desirability bias) because they were afraid that this would influence their grades and were thus trying to please the instructors. We addressed this threat by collecting the responses anonymously and by preventing multiple responses from the same student.

Novelty bias is an additional validity threat. The fact that most students interact with software theater the first time in their studies could cause an increased interest. We think that this threat is low because students who participate in the capstone course twice are equally motivated about software theater during their second time. To alleviate the threat of selection bias, we asked students in which team they worked: in each team, at least half of the team members responded in the online survey. We also analyzed the results on a team basis. While there are small deviations between the different teams, the general positive opinion about software theater is present in all teams. In addition, we found similar results in informal, personal interviews that support the statements we found in the questionnaire. Therefore, we think that the threat of selection bias is low.

Another threat to the validity in the evaluation is that most students were beginners in the taught concepts and mostly reported about their perceived experience within the course. Beginners

Table 2. Overview of the Schedule and Content in the Course POM

Week	Content
1	Team formation
2	Project organization
3	Software process models
4	Agile methods
5	Prototyping
6	Proposal management
7	Branch & merge management (Krusche et al. 2016)
8	Contracting
9	Continuous integration
10	Continuous delivery (Krusche and Alperowitz 2014)
11	Software theater
12	Global project management (Li et al. 2016)
13	Project management antipatterns

might not be able to objectively generalize their experience to other situations. Other variables of the course, such as an open atmosphere toward feedback, can have a positive influence on the evaluation result. If a student likes the capstone course, it does not necessarily mean that software theater was helpful. We were not able to exclude these variables in the evaluations of the case study. To alleviate these threats, we also analyzed how the students used software theater and these results support the findings of our study.

Our findings apply to a multi-customer software engineering course that was set up at our university. In other universities with different curricula and environments, it might not be possible to instantiate our course format and apply the software theater workflow easily.

7 CASE STUDY 2: INTERACTIVE LECTURE-BASED COURSE

In 2015 and 2016, we applied software theater in the university course "Software Engineering II: **P**roject **O**rganization and **M**anagement" (POM). We taught the course POM in the summer semester of 2015 with 294 students and in the summer semester of 2016 with 272 students who completed the final exam. 200 students regularly attended class and participated in exercises. Two distinct groups participated in the course: (1) bachelor students in information science, a few with experience in software engineering, and (2) master students in computer science, some with existing experience in the taught topics. The challenge of this heterogeneity was that students had different prior knowledge and completed in-class exercises at different speeds.

7.1 Course Description

The course POM has the following learning goals: participants understand the key concepts of software project management, in particular agile methodologies. They are able to deal with problems such as writing a software project management plan, initiating and managing a software project, and tailoring a software lifecycle. They are familiar with risk management, scheduling, planning, quality management, and build and release management. They can apply different techniques to solve development and management problems. Table 2 shows the schedule and the content of the lecture.

The course is based on active learning to increase student involvement and excitement with the subject being taught (Bonwell and Eison 1991). It integrates individual and team exercises into

the lectures (Krusche et al. 2017). Students participate in a team project with five team members, a simplified version of the projects in our capstone course (cf. Section 6). The goal of the project is to experience and apply the learned concepts in a more realistic environment. The instructor played the role of the customer and provided three short problem statements about the development of mobile applications. The teams had to choose one of the problem statements and one mobile app development environment: Android, iOS, or Xamarin.

Students formed teams with five team members on their own in the first lecture, following requirements to create balanced teams with respect to experience, nationality, and gender. In 2015, the students formed 58 teams, and in 2016, the students formed 51 teams. Software engineering is a collaborative activity (Whitehead 2007); therefore, team work is an important development skill students have to learn in the course. The teams used Rugby (Krusche et al. 2014) as an agile and continuous process model with an initial warm-up phase and five development sprints of 2 weeks each.

While individual in-class exercises are described in detail so that students can follow step by step, students receive a more vague description of the exercises in their team project that deliberately misses detailed instructions. The teams have to bring in their own ideas on how to solve the team project and apply the techniques they learned in individual exercises earlier in the course. For instance, we show them how to create low-fidelity prototypes with the tool Balsamiq in a detailed step-by-step tutorial in the individual in-class exercise. The corresponding exercise in the team project intentionally only states that the team should create prototypes: they can then choose their own preferred tool and tailor the prototype creation to their specific problem statement.

We introduced software theater in the 11th week of the course. In the class, we introduced students to the workflow and they applied the first steps directly in class within the context of their team project. They wrote the initial version of the demo scenario and the screenplay. As homework, they refined these artifacts, filmed the demo using their smartphones, and then uploaded a recording to the learning management system so that instructors and teaching assistants could review it.

7.2 Evaluation Design

We investigated the students' improvements in the exercises using an optional online questionnaire. We also asked them about their opinion on the exercise concept. It included questions about personal data, the participation in individual exercises, and application of techniques in the team project. We wanted to know if students improved their skills in software theater and if they felt confident to apply software theater in their next team project. Students could also comment on how the course can be improved.

We conducted the survey in July 2016 and gave the students of the 2016 course 2 weeks to complete it. Personalized tokens allowed the 272 students who completed the final exam of the course to participate exactly once in the anonymous survey.[13] We received 190 responses, which amounts to a response rate of 70%. In addition, we evaluated the use of software theater in 2015 and 2016 and analyzed the software theater artifacts of all teams who participated in the exercise. We counted how many teams delivered certain artifacts and how the teams scored in the exercise.

7.3 Evaluation Results

We analyzed the results of the online questionnaire and the uploaded artifacts of all teams who participated in the exercises. In the online questionnaire, 48% of the survey respondents stated

[13]The open source survey tool LimeSurvey (http://www.limesurvey.org) guarantees that the answers are anonymous by strictly separating token and answer tables in the database.

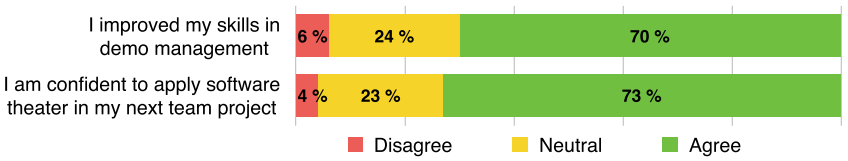

Fig. 21. Survey responses: disagreement (red) and agreement (green) with perceived improvements and confidence in software theater on a 3-point Likert scale.

Table 3. Number of Software Theater Artifacts Uploaded by the Participating Teams in the Lecture-based Course

Year	Teams	Formalized scenario	Screenplay	UML component diagram	UML class diagram	Demo backlog	Video	Feedback
2015	58	45 (78%)	39 (67%)	18 (31%)	24 (41%)	31 (53%)	20 (34%)	-[14]
2016	51	44 (86%)	43 (84%)	16 (31%)	23 (45%)	27 (53%)	20 (39%)	12 (24%)

that they participated in the in-class exercise about software theater **and** used it in their team project. Figure 21 shows that from these exercise participants, 70% agree that they were able to improve their skills in demo management (including software theater) and 73% agree that they were confident to apply software theater in their next team project.

In the statements about the exercise concept, students stated that software theater integrates creativity, interactivity, and fun into a lecture-based course, which is not common to most other, usually more static, lectures. The creation of a software theater demo improved their personal learning experience about software development and project management.

We evaluated how many teams participated in the software theater exercise and created the artifacts of the software theater workflow in both course instances in 2015 and 2016. The results are shown in Table 3.

The numbers in both courses in 2015 and 2016 were similar with a slightly higher participation in 2016. More than 80 teams participated in the software theater exercise: 78% of the teams in 2015 and 86% of the teams in 2016 created a formalized scenario; 67% (2015) and 84% (2016) created a screenplay. 53% of the teams created a demo backlog. Less than half of the teams created and uploaded UML diagrams. In both years, more teams created UML class diagrams than UML component diagrams. In total, 40 teams (34% in 2015 and 39% in 2015) performed the software theater demo, recorded a video, and uploaded it; 12 teams (24%) obtained feedback from other students in 2016. In 2015, we did not ask the students to collect feedback. Eleven teams (22%) in 2015 and seven teams (14%) in 2016 were able to complete all artifacts. We discuss these findings in Section 8.

7.4 Threats to Validity

Most of the limitations mentioned in Section 6.4 also apply for the lecture-based course, because we used similar evaluation techniques. We addressed social desirability bias by collecting the responses anonymously and by preventing multiple responses from the same student. Other variables of the course, such as the high rate of interactivity or an open atmosphere toward feedback, can have a positive influence on the evaluation result. If a student likes the lecture-based course, it does not necessarily mean that software theater was helpful. We were not able to control these

[14]We did not ask the teams in 2015 to collect feedback.

Table 4. Findings of the Evaluations in Relation to the Stated Hypotheses

ID	Hypothesis	Result
H1	**Increased motivation**: software theater increases the motivation of students to prepare a demonstration already early in the project.	Supported
H2	**Higher creativity**: demonstrations using software theater are more creative than normal demonstrations (without software theater).	Supported

variables in the evaluations of the case study. However, the evaluation results indicate that software theater can also be taught successfully in a lecture-based course.

8 DISCUSSION

In this section, we discuss the results of the two case studies with regard to our hypotheses mentioned in Section 1. We also provide best practices for other instructors who want to teach software theater.

8.1 Findings

Our case studies revealed that software theater can be taught to students in different types of courses. We found anecdotal evidence that software theater increases fun, motivation, and creativity in education, and is a technique to make software demos more relatable. Table 4 summarizes that our findings support the two proposed hypotheses: software theater increases the motivation of students to prepare a demonstration already early in the project (H1) and demonstrations using software theater are more creative than demonstrations without the technique (H2).

The hypothesis H1 is supported by our measures, both in the capstone course and the lecture-based course. While we encourage live demonstration in the design review of the capstone course and required it for the client acceptance test, we did not demand the usage of software theater for the demo. Nevertheless, 9 out of 11 teams used it to demonstrate a very early software prototype at the first event, and all teams used it in their final presentation. Although most teams could only show a rough first version of their system, their demo took up almost 25% of their maximum presentation time of only 10 minutes, and 24% of survey respondents spent 10 hours or more preparing it. This time investment shows recognition for the importance of a live demonstration using software theater at an early stage. However, as we do not have a control group that never learned about software theater, a further case study would be needed to fully examine this hypothesis.

In our lecture-based course, more than 70% of the students who applied software theater reported in the online questionnaire that they improved their skills in demo management and they are confident to apply software theater in their next team project. The number of participating students in the software theater exercise was relatively high, considering that it was one of the last exercises in the semester and required high effort. In both lecture-based courses in 2015 and 2016, more than a third of all teams performed the software theater and uploaded a video of their performance.

Hypothesis H2—software theater makes demonstrations more creative—is supported both by the survey and our observations during the case studies. 87% of respondents think that a demonstration with software theater is more creative than without, and the majority of them would employ the approach in future projects. We also observed a wide variety of creative techniques used in the demonstrations, such as the explanation of complex technological concepts with a narrator, through the theater play itself, and the use of metaphors from other domains to illustrate

the requirements of a system. Some teams also interweaved their demo with the presentation to support their arguments with a representation of their system in the usage context. While we do not have sufficient empirical data to back up this hypothesis, discussions with coaches, developers, and clients led us to the conclusions that software theater is a creative technique to demonstrate software systems, including early prototypes.

In addition, we observed that software theater demos help us in finding new customers for the capstone courses in the following semesters. We show potential customers the most creative parts and provide examples of what they can expect from the students. Customers can use the videos in their company to promote the collaboration with the university. The videos convince the industry partners about the application-oriented teaching and research at our department.

Another interesting aspect is that students share the software theater videos with family and friends to explain to them in an easy way what they achieved in their field of study. The students report that their relatives and friends without information technology background also understand how the developed software systems can help to improve specific problems. Software theater then generates a shared understanding between people with and without an information technology background and has a social relevance to facilitate digitalization and innovation. However, we do not have empirical evidence about this aspect yet.

Developers and team coaches agreed that software theater facilitates understanding of other student projects and increases confidence about their own projects in front of external parties. However, the opinions were divided about improved understanding concerning team-internal stakeholders, especially in communication with the client. We suspect that this is due to the setup of the capstone course: clients specify the project requirements in a problem statement and are usually strongly involved in the project: they regularly attend team meetings and review executable prototypes. With an already good knowledge of the requirements and up-to-date information about the current project state, a demonstration with software theater becomes less necessary because the added usage context does not provide further value. Thus, it is logical that teams do not prepare a demonstration using software theater for typical client meetings, but resort to a normal demonstration, which requires less preparation. In order to explore the improvement of understanding further, more data is needed on stakeholders who are less involved in the project, e.g., clients from other departments of the organization.

If a team chose to use software theater for their demo, we did not require them to go through every step of the workflow, but merely provided them with examples for each step and gave them the freedom to deviate. A majority of teams produced a screenplay for both events of the capstone course and the proportion of teams to produce a formalized demo scenario and a demo backlog increased between design review and client acceptance test. This implies that the students realized the usefulness of these artifacts for the software theater workflow and created them by choice because they helped them in their demo preparation. Over 70% of respondents indicated that they mocked parts of their system for the demonstration and the reasons stated for using mocks correspond to the recommendations we communicated during the lectures. Therefore, we can conclude that our teaching approach combining lectures, team coaching, and regular feedback in team sessions and dry runs leads to an improved understanding of the software theater workflow.

8.2 Best Practices

Having taught software theater in capstone courses since 2012 and in a lecture-based course since 2015, we have iteratively refined both our workflow and the approach of teaching it. We want to share the most relevant learnings for educators who want to adopt the method in their own courses.

Communicate knowledge multiple times: Our courses run over 3 months and students have to absorb a large amount of information in a relatively short amount of time. We aim to communicate the most important topics several times and through varying channels to ensure that everyone understands it correctly. In our capstone course, for instance, we give a general overview over software theater in a lecture, in which we combine theory with hands-on examples. The team coaches reiterate the workflow again, concentrating on the aspects that are most important for their specific team. We also revisit parts of the workflow in the cross-project teams and give feedback to each team how they apply software theater in their dry run before the real presentation.

Emphasize iteration: Students need multiple iterations to make sure that their demonstration represents their system and integrates nicely with the rest of their story in their presentation. Therefore, we introduce software theater at least three weeks before they first need to perform a demonstration, giving the teams enough time to prepare. We encourage them to go through multiple iterations of their screenplay. In our capstone course, we also hold a dry run in the week before each main event to give feedback to demo and presentation. To reduce the workload of the instructors, the team coaches help in the preparations and give feedback multiple times. The teams also help each other by commenting on others' presentations and demonstrations.

When it comes to the dry runs for the design review, our first event in the capstone course, it is common that teams receive challenging feedback from the instructors leading to fundamental changes in their demo. Our experience shows that the dry runs for the client acceptance test, our second event, require less input from instructor side. We believe that our course participants learn most about software theater by experiencing it in the first presentation. Therefore, we would recommend to give students at least two opportunities to perform a live demonstration during a course and to introduce mechanisms of peer feedback to keep instructor workload manageable.

Encourage creativity: It is helpful to show several examples of the application of software theater and to include cases in which a technical detail is explained (cf. Section 5.2), or where a team uses concepts from other fields to explain an abstract concept (cf. Section 5.3). We found that this not only shows students what is possible, but also motivates them to get creative and imagine how software theater would benefit the demonstration of their own system.

Stress the importance of preparation: In a demonstration, a lot of things can go wrong. We make sure to show common points of failure (e.g., a failed network connection) and to stress the importance of having a fallback. Most of the teams decide to mock parts of their system and prepare backup slides showing a video recording of the demo in case something goes wrong.

Document learnings: According to our experience, delivering a demo with software theater leads to an improved understanding of the technique itself. We encourage our student teams to perform a retrospective after their first presentation, discussing and documenting how it went and what they would like to improve. Changes in the adoption of software theater between the design review and the client acceptance test (e.g., more preparation time, more integrated demonstrations) shown in Section 6.3 are based on these learnings.

9 RELATED WORK

In the beginning of the 1990s, Brenda Laurel presented the idea that usability alone would not be enough in the development of successful computer systems (Laurel 1993). She described a theory of interaction, combining her experience in human computer interaction with knowledge of theater. "The real issue," she claims, is "How can people participate as agents within representational contexts? Actors know a lot about that, and so do children playing make-believe." Her hypothesis is that the vocabulary of traditional theater is similar to the vocabulary in human computer interaction. Laurel's work is based on Aristotle's poetics for dramatic theory, and explains how concepts such as catharsis, engagement, and agency can be applied in digital contexts.

The users' enjoyment must be a design consideration, which requires an awareness of dramatic theory and technique. She claims that effective design of interactive systems, like effective drama, must engage the users directly in an experience involving both thought and emotion: "Even in task-oriented applications, there is more to the experience than getting something done in the real world, and this is the heart of the dramatic theory of human-computer interaction." Our approach of software theater is based on Laurel's ideas and introduces theater into software engineering education to teach students the importance of interaction and emotion.

Mahaux and Maiden (2008) and Mahaux et al. (2010) proposed improvisational theater to support team-based innovation in the requirements engineering process. The commonality of their improvisational theater and software theater is that they both employ the form of theater as an effort to improve stakeholder communication and increase mutual understanding. But they differ in several aspects.

First, the purpose of improvisational theater is to generate creative ideas in the requirements engineering process, while the purpose of software theater is to demonstrate and evaluate design ideas for innovative software projects in education. Second, improvisational theater, as its name suggests, takes advantage of unplanned improvisational performance to stimulate the creativity of team members, while software theater emphasizes a pre-defined screenplay to set a framework for the demonstration. Third, software theater presents not only the applicability of user requirements, but also the feasibility of system requirements such as architecture design and hardware performance. Software theater is used in combination with specific software process and prototyping techniques, in our case the Tornado model. The Class, Responsibilities, and Collaborators (CRC) cards were introduced by XP practitioners Kent Beck and Ward Cunningham in 1989 as a teaching tool for object-oriented programming (Beck and Cunningham 1989). Instructors have used CRC cards for role-playing: each student plays the role of a specific class. They interact with other students representing collaborating classes to understand their responsibilities. While software theater is based on actors taking one or more roles, it is not designed to teach role-oriented programming (Kristensen 1995). Its purpose is to teach the demonstration of a system developed during a course. Rice and his colleagues used forum theater (a kind of interactive theater) to elicit requirements in the development of new technologies (Rice et al. 2007). They discovered the usefulness of storytelling through theater and video in promoting user involvement because it is easier for users (e.g., elderly people) that are not familiar with the state-of-the-art technologies to understand the system. They conclude that the technique "can increase designer empathy towards end users."

10 CONCLUSION

Software engineering is creative, imaginative, and interactive. However, simple demonstrations of the functional and static aspects of software do not provide the real-world usage context that is integral to understand the software requirements. We therefore advocate for a more dynamic way of presenting software prototypes. In this article, we described software theater, a combination of agile methods, scenario-based design, and theatrical aspects, as a way to demonstrate visionary scenarios in a more relatable and creative way, even if the software is not yet fully realized. The creation of the demo is based on the Tornado model and allows the validation of requirements, design, and technology decisions. We explained the software theater workflow in detail, including all steps from the customer's visionary scenario to the developers' actual demonstration.

We illustrated the artifacts of one example and showed three common software theater patterns. We taught software theater in a capstone course with industrial customers and in an interactive lecture-based course, described both courses as case studies, and explained our teaching approach. In empirical evaluations in these two courses, we found that software theater motivates

the students to prepare demonstrations even early in the project. Students acquire the ability to apply software theater within one semester and perceive software theater demonstrations as more creative, more memorable, more dynamic, and more engaging than normal techniques. While we found first anecdotal evidence that software theater improves the understanding of external stakeholders about the developed system, we further need to investigate this aspect.

Rise and his colleagues pointed out that theater "may not be equally well suited for very early requirements gathering, or later, more specific prototype evaluation" (Rice et al. 2007). We hypothesize that software theater can also be used as requirements elicitation technique and want to evaluate this in the future. We started to use it in other course formats such as short summer schools (1–2 weeks), but have not yet collected data about its usage and benefits in these shorter courses. Another research area is the further formalization of the described software theater patterns as well as the identification of further patterns. We want to use a common scheme such as the one proposed by the Gang of Four (Gamma et al. 1995) to describe different educational software theater patterns including their motivation, applicability, context, forces, and consequences.

ACKNOWLEDGMENTS

We want to thank all students of the capstone courses and the interactive lecture-based courses. We also thank our colleagues for management support, filming, and technical support and our industry partners for providing interesting problem statements.

REFERENCES

Russell Abbott. 1983. Program design by informal English descriptions. *Commun. ACM* 26, 11 (1983), 882–894.

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Joaquim Romaguera i Ramió, Max Jacobson, and Ingrid Fiksdahl-King. 1977. *A Pattern Language*. Gustavo Gili.

Lukas Alperowitz. 2017. *ProCeeD—A Framework for Continuous Prototyping*. Ph.D. Dissertation. Technical University Munich, Germany.

Larry Apfelbaum and John Doyle. 1997. Model based testing. In *Proceedings of the Software Quality Week Conference*. 296–300.

Jonathan Arnowitz, Michael Arent, and Nevin Berger. 2010. *Effective Prototyping for Software Makers*. Morgan Kaufmann.

Victor Basili. 1996. The role of experimentation in software engineering: Past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE, 442–449.

K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, and others. 2001. Manifesto for agile software development. *The Agile Alliance* (2001). http://agilemanifesto.org.

Kent Beck and Ward Cunningham. 1989. A laboratory for teaching object oriented thinking. In *Sigplan Notices*, Vol. 24. ACM, 1–6.

Barry Boehm. 2000. Requirements that handle IKIWISI, COTS, and rapid change. *Computer* 33, 7 (2000), 99–102.

Charles Bonwell and James Eison. 1991. *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Reports.

Bernd Bruegge and Allen Dutoit. 2009. *Object Oriented Software Engineering Using UML, Patterns, and Java* (3rd ed.). Prentice Hall.

Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. 2015. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education* 15, 4 (2015), 17:1–17:31.

Bernd Bruegge, Stephan Krusche, and Martin Wagner. 2012. Teaching tornado: From communication models to releases. In *Proceedings of the 8th Edition of the Educators' Symposium*. ACM, 5–12.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. A system of patterns: Pattern-oriented software architecture. Addison Wesley.

Lan Cao and Balasubramaniam Ramesh. 2008. Agile requirements engineering practices: An empirical study. *IEEE Software* 25, 1 (2008), 60–67.

John Carroll. 1995. *Scenario-based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, Inc.

John Carroll. 2000. *Making Use: Scenario-based Design of Human-computer Interactions*. MIT Press.

Dora Dzvonyar, Stephan Krusche, and Lukas Alperowitz. 2014. Real projects with informal models. In *Proceedings of the 10th Edition of the Educators' Symposium*.

Christiane Floyd. 1984. A systematic look at prototyping. In *Approaches to Prototyping*. Springer, 1–18.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

Ron Garland. 1991. The mid-point on a rating scale: Is it desirable. *Marketing Bulletin* 2, 1 (1991), 66–70.

Orit Hazzan. 2002. The reflective practitioner perspective in software engineering education. *Journal of Systems and Software* 63, 3 (2002), 161–171.

Matthias Jarke, Ralf Klamma, Klaus Pohl, and Ernst Sikora. 2010. Requirements engineering in complex domains. *Graph Transformations and Model-driven Engineering* (2010), 602–620. http://dblp.uni-trier.de/rec/bibtex/conf/birthday/JarkeKPS10.

Matthias Jarke and Klaus Pohl. 1993. Establishing visions in context: Towards a model of requirements processes. In *Proceedings of the 14th ICIS*.

Stephen Kline and Nathan Rosenberg. 1986. An overview of innovation. In *The Positive Sum Strategy: Harnessing Technology for Economic Growth*. National Academy Press, 275–305.

Bent Bruun Kristensen. 1995. Object-oriented modelling with roles. In *OOIS*. 57–71.

Stephan Krusche. 2016. *Rugby - A Process Model for Continuous Software Engineering*. Ph.D. Dissertation. Technical University Munich, Germany.

Stephan Krusche and Lukas Alperowitz. 2014. Introduction of continuous delivery in multi-customer project courses. In *Proceedings of the 36th International Conference on Software Engineering*. IEEE, 335–343.

Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin Wagner. 2014. Rugby: An agile process model based on continuous delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 42–50.

Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. 2016. Teaching code review management using branch based workflows. In *Companion Proceedings of the 38th International Conference on Software Engineering*. IEEE, 384–393.

Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge. 2017. Interactive learning: Increasing student participation through shorter exercise cycles. In *Proceedings of the 19th Australasian Computing Education Conference*. ACM, 17–26.

Brenda Laurel. 1993. *Computers as Theatre* (2nd ed.). Addison-Wesley.

Meir Lehman and Laszlo Belady. 1985. *Program Evolution: Processes of Software Change*. Academic Press.

Yang Li, Stephan Krusche, Christian Lescher, and Bernd Bruegge. 2016. Teaching global software engineering by simulating a global project in the classroom. In *Proceedings of the 47th SIGCSE*. ACM, 187–192.

Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-testing: Unit testing with mock objects. In *Extreme Programming Examined*. ACM, 287–301.

Martin Mahaux, Patrick Heymans, and Neil Maiden. 2010. Making it all up: Getting in on the act to improvise creative requirements. In *Proceedings of the 18th International Requirements Engineering Conference*. IEEE, 375–376.

Martin Mahaux and Neil Maiden. 2008. Theater improvisers know the requirements game. *IEEE Software* 25, 5 (2008), 68.

Robert Martin. 1996. The dependency inversion principle. *C++ Report* 8, 6 (1996), 61–66.

Jakob Nielsen. 1994. *Usability Engineering*. Elsevier.

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

Donald Norman and Stephen Draper. 1986. User centered system design: New perspectives on human-computer interaction. CRC Press.

Tom Nurkkala and Stefan Brandle. 2011. Software studio: Teaching professional software engineering. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM, 153–158.

Klaus Pohl. 2010. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer.

Roger Pressman. 2009. *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Mark Rice, Alan Newell, and Maggie Morgan. 2007. Forum theatre as a requirements gathering methodology in the design of a home telecommunication system for older adults. *Behaviour & Information Technology* 26, 4 (2007), 323–331.

Colette Rolland, C. Ben Achour, Corine Cauvet, Jolita Ralyté, Alistair Sutcliffe, Neil Maiden, Matthias Jarke, Peter Haumer, Klaus Pohl, Eric Dubois, and P. Heymans. 1998. A proposal for a scenario classification framework. *Requirements Engineering* 3, 1 (1998), 23–47.

Ken Schwaber and Mike Beedle. 2002. *Agile Software Development with Scrum*. Prentice Hall.

Helen Sharp. 2003. *Interaction Design*. John Wiley & Sons.

Mary Shaw, Bernd Bruegge, and John Cheng. 1991. A software engineering project course with a real client. Carnegie Mellon University Pittsburgh Software Engineering Institute.

Mary Shaw, Jim Herbsleb, Ipek Ozkaya, and Dave Root. 2006. Deciding what to design: Closing a gap in software engineering education. 28–58.

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

James Tomayko. 1987. *Teaching a Project-intensive Introduction to Software Engineering*. Technical Report CMU/SEI-87-TR-20. DTIC Document.

Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. 1998. Scenarios in system development: Current practice. *IEEE Software* 15, 2 (1998), 34–45.

Jim Whitehead. 2007. Collaboration in software engineering: A roadmap. *FOSE* 7, 214–225.

Han Xu, Oliver Creighton, Naoufel Boulila, and Bernd Bruegge. 2013. User model and system model: The yin and yang in user-centered software development. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. ACM, 91–100.

Han Xu, Stephan Krusche, and Bernd Bruegge. 2015. Using software theater for the demonstration of innovative ubiquitous applications. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 894–897.

Received April 2017; accepted August 2017