Teaching Tornado

From Communication Models to Releases

Bernd Bruegge
Department of Computer Science, Technische Universitaet Muenchen
bruegge@in.tum.de

Stephan Krusche
Department of Computer Science, Technische Universitaet Muenchen
krusche@in.tum.de

Martin Wagner
Department of Computer Science, Technische Universitaet Muenchen
wagmarti@in.tum.de

ABSTRACT

In this paper we describe Tornado, which we teach in our software engineering project courses. Tornado is a new process model that combines the Unified Process with Scrum elements.

The Tornado model focuses on scenario-based design starting with visionary scenarios funneling down to demo scenarios. Tornado offers models for a broad range of activities. In addition to formal models used for analysis and design, Tornado encourages the developer to use informal models as communication medium for the interaction with the customer and end user. These communication models can be used as the basis of early releases to increase the feedback from customer to developer. We argue that the combination of informal modeling and release management can be introduced early in software engineering project courses.

We describe a case study in which we demonstrate the use of communication models and release management in a multi-customer course with 80 students. In three months, the students produced 163 releases for 11 customers.

Categories and Subject Descriptors

K.6.3 [**Management Of Computing And Information Systems**]: Software Management—*software development, software selection*; D.2.9 [**Software Engineering**]: Management—*software configuration management, software process models*

General Terms

Management, Design, Human Factors

Keywords

Agile Techniques, Continuous Integration, Executable Prototypes, Extreme Programming, Scenario-Based Design, Informal Modeling, Project Courses, Scrum, Prototyping, Release Management, Software Engineering Education, Unified Process

.

1. INTRODUCTION

There is an acknowledged need for teaching modeling in project intensive courses for all Computer Science majors, not just Software Engineers. One of the drawbacks of project courses is that instructors cannot escape the complexity and change that their students experience. Instructors quickly become participants in the development themselves, often acting as project managers. In other words, the challenge for a non-trivial project course is how to make the project complex enough to enrich the students software engineering experience, yet simple enough to have a feasible solution that does not unduly burden them or the instructor.

Model-driven development is a trend which has slowly moved into many Software Engineering courses. However, many customers are not yet knowledgeable enough to buy into the model-driven approach. In addition, it is difficult to get time commitments from real customers. As a result, the instructor often plays the role of the customer. We think that clearly, this is not a good solution.

The main goal of any project course should be to teach about realistic situations. Instructors cannot expect to teach students how to deal with complexity and change if they have to simultaneously play the role of the customer, the project manager and - at the end of the course - the acceptance of the system. One way to simplify the project for instructors as well as students is to work in a problem domain that students themselves already know, or one they are motivated to learn more about (e.g., computer game development, or enhancing tools oriented toward program development itself). One pitfall is that students often have a user's extensive knowledge of the game or tool interface but are blissfully unaware of what goes into modeling, implementation, testing and delivery.

We have been searching for the best way to teach realistic software engineering project courses starting in the 1980's [6], [22]. Since then, we have experimented with many different set-ups and parameters, and we are still experimenting. However, 3 parameters have stayed constant: We always look for a real customer who has a real problem to be solved and we ask the students to solve the problem by a real deadline, usually by the end of the semester. Even with these constraints, there is a wide spectrum of possibilities to teach a project course determined by several factors.

A problem-independent way that students can be exposed to the complexity of real projects is to have them work together in larger teams where they experience realistic communication problems. If the teams are diverse, they can exploit synergy from disparate backgrounds. The complex-

ity of the course can also be increased by the number of problems to be solved, and the number of customers.

Our courses stress modeling in its various forms, ranging from informal models to UML models. Our early courses have dealt mostly with modeling desktop-oriented information systems, where we focused on system modeling, in particular object modeling, functional modeling and dynamic modeling [7]. With the emergence of smartphones as a new exciting platform, we are focusing more and more on the development of mobile interactive systems. This requires additional modeling activities, in particular, we now also include user modeling, user interface modeling and usability testing in our courses.

All the courses we have taught can be placed into four different categories: *Single-Project* courses, *Global SE* courses, *Multi-Project* courses and *Multi-Customer* courses (see Table 1).

Course	Course Type	# TAs	# Students	# Teams	# Locations	# Customers	# Problems
IP	Single-Project	3	30	5	1	1	1
JAMES	Global SE	5	110	8	2	1	1
DOLLI 5	Multi-Project	5	47	6	1	1	2
iPraktikum	Multi-Customer	11	80	11	1	11	11

Table 1: Examples of our project courses

A single-project course has a single customer stating a single problem to be solved by all the students working together in teams. In 1991, we taught Interactive Pittsburgh (IP), a single-project course with the Pittsburgh City Planning Department and 30 students [5]. A global software engineering course consists of one or more customers distributed across multiple locations. From 1997 to 1998, Daimler in Stuttgart and Chrysler in Detroit acted as customers in the JAMES project with 110 students working in teams at two universities, Carnegie Mellon University and Technische Universitaet Muenchen [11]. A multi-project course is a course with a single customer requesting several problems to be solved by the students. In the last six years we worked on the DOLLI project with the Munich Airport as the customer in several multi-project courses [8], [9].

The most ambitious instance of the project courses is the multi-customer course, which requires a highly motivated instructor. A multi-customer course is a course with more than one customer and with many problems, each of them to be solved by one team of students. In the summer of 2012 we organized iPraktikum, a multi-customer course with 80 students working on 11 problems from 11 companies [12].

The usual reaction from many instructors we have talked to is that teaching such a course is not possible. The purpose of this paper is to show that it is indeed possible, in particular with the recent advances in continuous integration and the emergence of release management tools that can be used effectively in the class room.

The paper is organized as follows. Section 2 describes our software process which we call Tornado. Tornado is a hybrid model based on the Unified Process enhanced with many Agile concepts. In the pre-development phase the instructor and the customer determine the scope of the problem and the requirements, and describe them with a set of visionary scenarios and a top-level design. These are presented to the students at the beginning of the course.

Section 3 describes the modeling activities of the development phase and the various types of models which the students are required to learn and use, ranging from informal models to formal models. We expect the students to read chapters of our text book in parallel, in particular the chapters covering modeling.

In the weekly meetings, we show how these models can then be used for the problem to be solved. Quite early we stress the importance of changing requirements. The customer prioritizes the visionary scenarios and students develop them using vertical integration. Each vertical slice leads to a *touchpoint*, an executable prototype which can be sent to and immediately used by the customer.

In section 4 we describe our release management process which enables us to deliver these touchpoints much earlier and much more frequently in the project than was possible even a few years ago. Section 5 describes our experience with a multi-customer course with 80 students building 16 applications for 11 customers, successfully producing 163 releases during a single semester.

2. THE SOFTWARE PROCESS

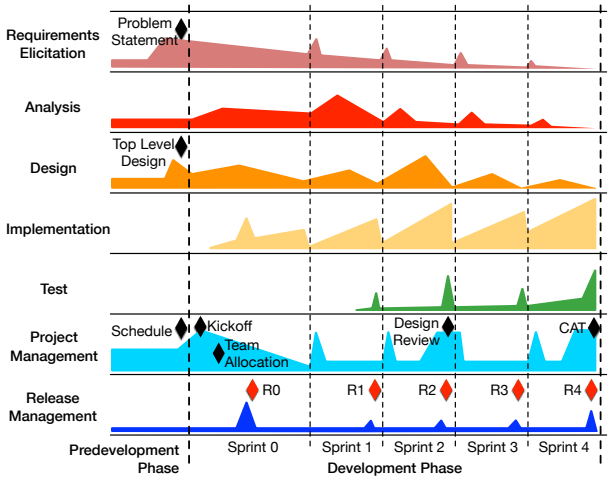


Figure 1: The course lifecycle model (adapted from [14])

Figure 1 illustrates the process used in our courses. It is based on the Unified Process model [14] enhanced by Agile constructs, in particular Scrum [20] and XP (Extreme Programming)[2]. During the *Pre-Development Phase*, the instructor identifies the customers. Typically this should be a couple of months before the *Kickoff*, but we have experienced cases where the customer was identified just a few days before the kickoff. During this phase, the identification of scenarios, functional as well as nonfunctional requirements is important. We also identify a first *Top-Level Design* in terms of hardware and software components and an initial *Schedule* illustrating the key milestones in the course, in particular the times for reviews and deliveries.

During the kickoff, the customer presents the *Problem Statement* which contains the initial requirements in terms of *Visionary Scenarios* (see Figure 2) and an initial subsystem decomposition which is the Top-Level Design. Imme-

diately after the kickoff, the students have to indicate their preferences for the projects. According to their preferences and experiences, we divide them into teams. After the *Team Allocation* we partition the development into several *Sprint*s [20]. The duration of sprints varies. The longest one, Sprint 0, takes about 4 weeks and focuses on team building (e.g. with an icebreaker) to get everybody comfortable with the project, the customer and the problem. This sprint includes workshops, tutorials, interviews with the customer, trips to the target environment and a first release. In the workshop the customer prioritizes the visionary scenarios to be selected for demonstration in the successive sprints.

The focus of the first release (R0 in Figure 1) is an executable system, in which every subsystem of the top-level design is integrated. The goal is to get the students familiar with the release management workflow. Another task of the teams is the creation of an executable mockup of the user interface. In each subsequent sprint, which usually last about 2 weeks, the team initially meets the customer for sprint planning and then focuses on realizing executable prototypes (called potentially shippable product increments in Scrum).

Scenario-based design [10] is our preferred way of modeling requirements. We use *User Stories* (described in XP) to split these requirements into manageable pieces. The responsibility of the instructor is to ensure that the scope of each sprint can be successfully implemented by the students. The teams are self-organizing in the way described by Takeuchi and Nonaka [21].

An important milestone after sprint 2 is the *Design Review* (see Figure 1) where we require the presence of the customer. In this review the students present executable prototypes implementing the design scenarios, which are a refinement of the visionary scenarios (see Figure 2).

Figure 2: Tornado model: Wide in analysis, narrow in implementation

We teach a system integration strategy called *Tornado Integration* (see Figure 3c). The purpose of Tornado integration is to focus on usability and to produce a prototype which can be executed by the user to quickly collect feedback. Each executable prototype represents a vertical slice, which is a touchpoint in our Tornado model. Tornado integration avoids the disadvantages of horizontal and *vertical* integration. Horizontal integration strategies such as bottom-up and top-down require simulations with stubs, drivers or mock objects. Because of its focus on a single functionality at a time, vertical integration does not fully address all aspects of usability in the initial slices. This is especially important because we teach the user-centered design approach described by Norman [18].

In addition we use the tornado metaphor to teach the difference between visionary, design and demo scenarios (see Figure 2). A tornado is wide in the clouds and narrows down until hits the earth at a touchpoint. The same applies for the activities in our lifecycle model. Some of the visionary scenarios may be not realizable and are thus not considered in the design scenarios. The demo scenarios are the refinement of the design scenarios which are delivered at the end of project. In that sense the *CAT* (Customer Acceptance Test) - the final presentation at the end of the semester - is an unpredictable touchpoint demonstrating the most important functionalities of the system realized by the students.

3. INFORMAL MODELING

While the focus in teaching software engineering has often been on formal models, we emphasize the use of informal models, where the focus is on communication with other developers and customers. These models can even be incorrect, which means they can contain contradictions. We call an informal - and possibly incorrect - model a **communication model** whereas a formal and correct one is called a **specification model**. In this section we explain the advantages of informal modeling and the reasons why we teach it in addition to formal modeling.

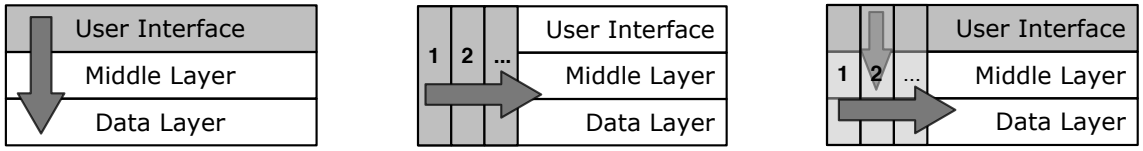
Historically software engineers have tried to use math-based modeling languages that allowed formal approaches and helped to verify software, but forgot the user. This is exemplified by Dijkstra's remark in an ICSE panel that "the notion of 'user' cannot be precisely defined, and therefore has no place in computer science or software engineering" [13]. We think the user plays a central role in software engineering.

The focus in formal modeling is to create models that are consistent, unambiguous, complete, correct, verifiable, and realizable. Specification models use the language of mathematics (Z, RSML, SCR, RML, etc). With a precise and unambiguous notation, discrete mathematics is applied to software engineering. Formal languages and formal reasoning is used to verify the correctness of the system. There are applications where the focus on specification models from the beginning makes sense, particularly in embedded systems where safety plays an important role.

But the exclusive usage of formal models is a problem when applied in the design or development of mobile interactive systems. Consistent and unambiguous models can hardly be achieved, especially when the customer does not yet completely know the requirements, which is always the case. Another problem of specification models is that they lead to analysis paralysis [4] because the developers try to describe all the requirements at the beginning of the project.

It is hard to describe the requirements for mobile, interactive, and usable systems up front because after their first experience, users usually ask for changes. In our courses, we therefore teach how to deal with incompleteness and change. Models can be incomplete, when developers need rapid feedback from the users, especially when they are in doubt about the usability of the system. In fact, a requirement might turn into the opposite as a result of an experience reported by the user.

In contrast to specification models, communication models can be incomplete, ambiguous, incorrect, unverified, and



(a) Top-Down (b) Vertical (c) Tornado Integration

Figure 3: Integration approaches

even unrealizable. Their main purpose is to enable and improve the communication about complex concepts between participants of a software project by providing an abstraction and simplification of the reality. Small syntax errors in models are allowed, we do not correct them right away, because students have often have to change them anyway.

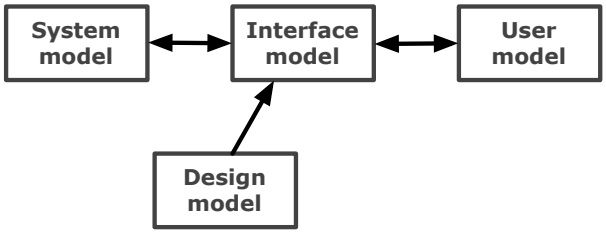


Figure 4: Models in User Interface Design (adapted from Norman and Draper [18])

We actually believe that informal modeling is a creative process (e.g. during brainstorming) that helps to overcome the gap between two different mental models. Developers often understand concepts of the system differently than customers or users. This has been well described by Norman and Draper [18] and is shown in Figure 4.

Developers implement the requirements in the system model that describes the functionality, the structure and the behavior of the system. The interface model describes how the system is presented to the user to hide complex details of the system model and to increase the usability of the system. The interface model is influenced by the design model which reflects the understanding of the developer as to how the user should interact with the system. The user model is the user's idea of how the system should work, which unfortunately, may be different from the design model.

The earlier the developer gets feedback with the help of informal models, the earlier he can determine whether his understanding of the system matches the users understanding. The purpose of informal modeling is to quickly enable a common understanding of the system by closing the gap between the design model and the user model. We teach different techniques that help in the reduction of this gap. While we prefer UML for creating the system model, there are many other techniques to create the interface model. Developers can choose the technique that fits their needs. Here are some examples of informal models that we teach to the students:

- Back of the Napkin Designs
- Whiteboard or Paper Scetches
- Low-Fidelity User Interfaces
- Storyboards
- Narrative Texts
- User Stories

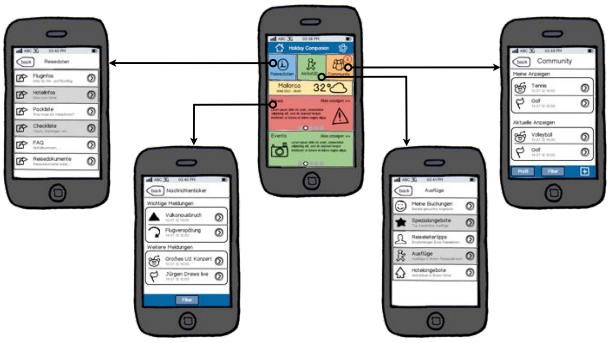


Figure 5: Low-fidelity user interface created with a mockup tool (Balsamiq)

There have been research results (e.g. by Rudd et al. [19] and Mayhew [16]) showing that unpolished user interfaces receive more feedback than polished ones. An example of an unpolished, low-fidelity user interface with screens and edges (created with Balsamiq[1]) is shown in Figure 5. We teach the students how to translate this informal model into a UML state diagram with nodes replacing the screens and transitions replacing the edges as shown in Figure 6.

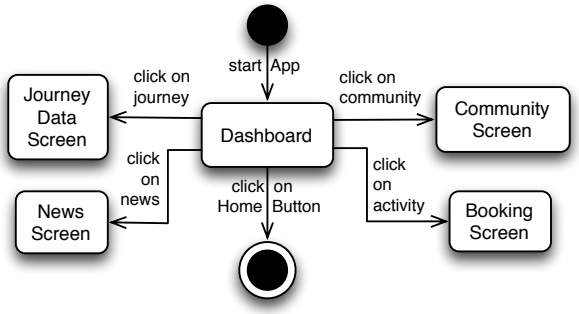


Figure 6: UML state diagram representing the user interface modeled in figure 5

An informal model does not follow formal rules and is not designed to be validated by a model checker, but focuses on showing the user how the user interface looks like and how to

[1]Balsamiq is an easy to use online mockup tool, see `http://www.balsamiq.com`.

obtain feedback about it. Figure 7 shows another example of a low-fidelity prototype, in this case the paper scetch of the user interface for an iPad game.

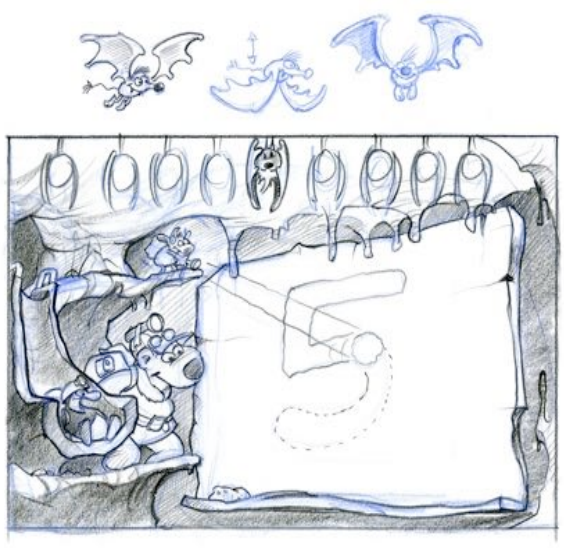


Figure 7: Example of a low fidelity interface

Low-fidelity prototypes are cheaper to produce and are easier to change than high-fidelity ones. For that reason they allow many more alternatives for comparison and more testing cycles. In fact, with the Tornado model, we can deliver executable prototypes with low-fidelity user interfaces to customers so that end users can perform usability tests [17]. In these tests, we receive more feedback than we would receive when we use the final user interface. An example of a low-fidelity usability test using the paper scetch of the user interface is shown in Figure 8.

Figure 8: Example of usability testing with a low-fidelity executable prototype

We believe the advantages of low-fidelity prototypes also apply to informal models. They are easy and quick to create and allow more iterations which leads to more feedback and the creation of more alternatives. Figure 9 shows an informal UML like model on a whiteboard, that follows no strict formal notation.

Informal modelers can create their own techniques as long

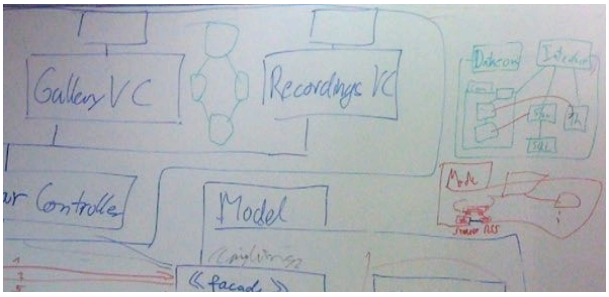


Figure 9: Informal UML model sketched on a whiteboard

as they are understood by other developers, users or customers. Moreover, developers can collaboratively create informal models and can even include the user or the customer into the modeling process to receive feedback [15]. Multiple iterations of informal models lead to faster results because changes are easy to make and the informality helps in understanding and communicating the system structure.

4. RELEASE MANAGEMENT

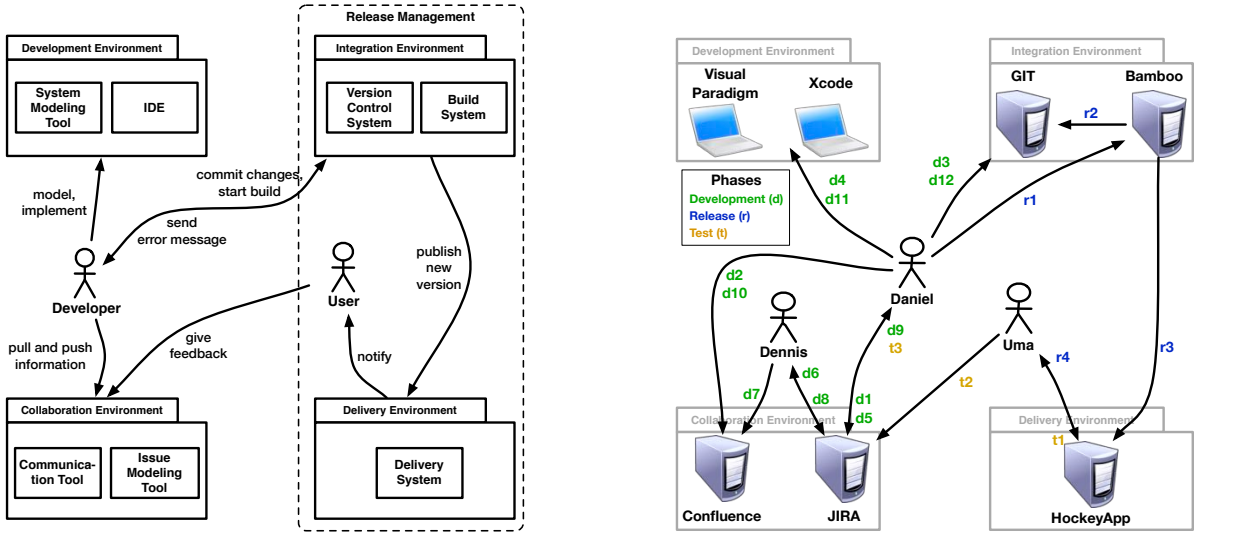
In this section we describe how the selection of the demo scenarios is supported by release management. We use external releases which are delivered to the customer and internal releases targeted to other students in the teams. Both release types are created by the release management process which is integrated into our course infrastructure. Figure 10a describes the 4 different environments of this infrastructure as well as the interaction of developers and users with it.

In our older courses, we asked the students to focus on the development of models and source code and interaction with each other. Acting as developers, they modeled the scenarios and implemented them with system modeling tools and IDEs (Integrated Development Environments). The discussion of the scenarios and the interaction with the user were supported by a *Collaboration Environment* using issue-based modeling and tracking as well as communication tools.

In the past we have tried many different tools. For the *Development Environment* we used commercial CASE tools such as OMTool, moving on to Enterprise Architect, Visual Paradigm or research CASE tools like UNICASE. For the collaboration environment we explored tools such as Lotus Notes, Bulletin Boards, Wikis, and even Email distribution lists.

None of these tools supported release management. With the emergence of tools supporting continuous integration and delivery, we have now been able to integrate two more environments into our course infrastructure. The *Integration Environment* consists of a version control system and a build system supporting continuous integration to create working versions of the software. The *Delivery Environment* enables teams to deliver the releases to the customers and to obtain feedback. In the following we describe our release management (the right part in Figure 10a) in more detail.

We require at least three releases in our courses. The first one, an internal release, is created on the basis of the subsystem decomposition right after the kickoff. Effectively this is a barebone system test. In Figure 1, this release is called R0. Informally we call it "Hello Dolly". Each subsystem provides a facade with one public method. The Hello Dolly test



(a) Project development infrastructure

(b) Example scenario for a release in the iPraktikum 2012 [12]

Figure 10: Release management integrated into the project course infrastructure

driver invokes these public methods and if successful, plays the melody from the musical. During Sprint 1 the students have to deliver the first scenario based touchpoint called R1 in Figure 1.

The next release (R2) is delivered to the customer at the design review milestone. Based on the customer's feedback, the students adjust the visionary and design scenarios[2]. The modified visionary scenarios are then the basis for the identification of the demo scenarios to be used in the release R4 for the CAT.

While creating the executable prototypes for the design review and customer acceptance test, the students typically increase their development activities. We have observed that shortly before a presentation students become increasingly ambitious, producing a multitude of releases, refining the scenarios, and implementing additional features. We have even observed this behavior with our customers, once they get used to the Agile approach.

This creates a conflict of interest between the demonstration of additional functionality and stability of the selected demo scenarios. We are now able to allow these conflicts, because our release management tools save previous releases including release notes and feedback. If the new demo release is stable and closer to the visionary scenarios, it will be selected for the presentation. As a fallback measure, the previous demo releases are available in our delivery server and can be reinstalled in case the newest release fails during the preparation for the demonstration. In those cases we tell the students about Murphy's law and advice them to prepare hardware software configurations for both demos. This can easily be achieved with our release management process.

5. CASE STUDY

We have used the course infrastructure in Figure 10a in our courses since 2010. In the following we describe an instance of a multi-customer course with 11 customers which we taught in the summer of 2012 [12]. The course was offered to 80 students who had to solve 11 problems, each laid out by one customer. We chose a matrix organization shown in figure 11 because of the number of projects. We moved the instructor to a new management layer, called program management, allowing for more than one instructor. The instructors were still responsible for interacting with the customers, in particular with respect to non-disclosure agreements. They set up the course infrastructure with a special focus on long term stability, while still supporting the lightness of agile development.

Each of the 11 projects was assigned to a development team (with 5-8 students) led by two people, a teaching assistant (usually a doctoral student) communicating with the program management and a coach supporting the teaching assistant. We selected coaches who were familiar with our course infrastructure and with the Tornado model: We recruited them from the student pool who had taken the course in a previous semester.

Additionally we formed 3 cross-project teams in the beginning of the course: a modeling team, a configuration management team and a code quality team. We had these types of teams already in earlier courses, but now we extended their tasks with respect to informal modeling and release management. The modeling team introduced informal modeling tools and helped in the transitions from communication to specification models. The configuration management team focused on continuous integration and release management. The code quality team performed regular reviews to improve the quality of the models and synchronize them with the source code. Each of the cross-project team members was also a member in a development team. This helped in the dissemination of knowledge established in the cross-project teams into the development teams without too much additional overhead for the instructors.

In the following we describe a release management sce-

[2]We use the term updraft based on our Tornado metaphor to describe the effect of the customers feedback on the visionary scenarios.

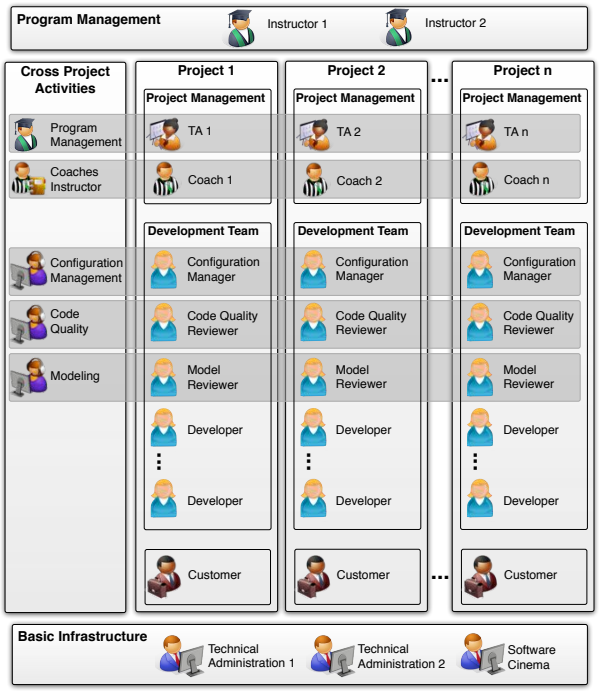
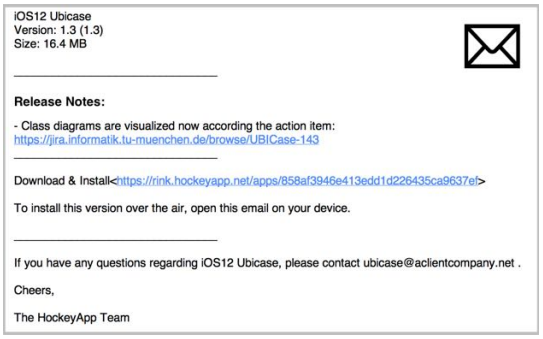


Figure 11: Organizational chart of the project course

nario (visualized in Figure 10b) to exemplify a typical workflow in our course infrastructure. The participating actors of the scenario are Daniel and Dennis, developers of the Ubicase team and Uma, tester and user employed by the Ubicase company. The involved systems are:

- JIRA [1] as issue modeling [7] tool
- Confluence [1] as communication tool
- Visual Paradigm for UML [23] as system modeling tool
- Xcode as integrated development environment (IDE)
- GIT as the version control system
- Bamboo [1] as build system
- HockeyApp [3] as delivery system

The scenario is divided into three phases: development, release and test. After a team meeting in the development phase Daniel takes a look at his action items in JIRA (d1). One of his action items is the development of a new use case, namely the visualization of class diagrams in an iPad application. To get an overview about his task he takes a look at the use case model and at the current subsystem decomposition, both stored in Confluence (d2). Before starting with his task he updates his working directory with the latest copy of the GIT repository (d3). First Daniel models the new use case and adds it into the use case model maintained with Visual Paradigm (d4). To implement the use case he starts Xcode (d4). But then he recognizes that the required graphic elements are not yet designed. As he needs them for completing his task, he creates a new action item in JIRA (d5) and assigns it to Dennis who is the graphics experts. Dennis receives a notification from JIRA and opens the action item (d6). He creates the new graphics, uploads them


iOS12 Ubicase
Version: 1.3 (1.3)
Size: 16.4 MB

Release Notes:

- Class diagrams are visualized now according the action item:
https://jira.informatik.tu-muenchen.de/browse/UBICase-143

Download & Install<https://rink.hockeyapp.net/apps/858af3946e413edd1d226435ca9637ef>

To install this version over the air, open this email on your device.

If you have any questions regarding iOS12 Ubicase, please contact ubicase@aclientcompany.net .

Cheers,

The HockeyApp Team

Figure 12: Notification Email from HockeyApp

to Confluence (d7), and resolves the action item (d8). Daniel is automatically informed by JIRA (d9) and downloads the new graphics from Confluence (d10). He is now able to implement the visualization use case with Xcode (d11). After unit testing his changes, he commits them into the GIT repository, using a tag in the commit message that indicates that he has finished his action item (d12). This is automatically picked up by JIRA which watches all the commit messages. As a result the action item is now resolved.

Now the release phase begins: Daniel starts a new build in Bamboo (r1). In Bamboo he sees a list of the last changes and a list of resolved action items in JIRA. He checks that his change is now part of the release and uses the information to write the release notes. In the release notes he includes links to the resolved action items in JIRA and explains the changes to the user. Bamboo automatically gets the most up-to-date version from the GIT repository (r2), compiles the source code and builds a new version of the iPad application. Bamboo then uploads the new version to HockeyApp (r3). HockeyApp sends a notification email including the release notes to all registered users (r4). The notification email is shown in figure 12.

Uma is one of the registered users. She receives the email (r4), clicks on the link to open HockeyApp and downloads the application to her iPad (t1). She plays with the new visualization feature and notices that class methods are not shown on her iPad. She finds a link in the release notes that allows her to place a comment in the action item in JIRA (t2). Daniel is automatically notified by JIRA about the new comment (t3). In the next team meeting the comment is discussed. The team decides that adding the methods is not difficult. The action item is reopened and Daniel promises to implement the suggested change. Now the release cycle starts again.

The described scenario shows one of many possible workflows in our course infrastructure. We also support workflows for the automatic upload of crash reports from mobile devices and the possibility for the user to directly give feedback in the executable prototype. The combination of JIRA and Confluence also allows us to simplify the creation of rationale-based meeting agendas [7].

In the iPraktikum course, the students developed and released 16 application to their customers. They created 2648 issues in JIRA, 2127 of them were resolved before the Customer Acceptance Test. The students committed their work more than 5500 times into the GIT repository. They created 831 builds in Bamboo, many of them potential releases. In

fact, 163 releases were delivered to our customers during the project time. This means, that on the average each application was delivered more than 10 times.

6. CONCLUSIONS

In this paper we described Tornado, which we use in our software engineering project courses, a process model that combines the unified process with agile techniques. Tornado focuses on scenario-based design starting with visionary scenarios narrowing down to demo scenarios. It combines the advantages of vertical and top-down integration. A touchpoint in the Tornado model is a successful delivery of an executable prototype. Updrafts provide early customer feedback and allow all participants to react to unpredictable changes.

We have made a case for informal modeling as a precursor to specification modeling. This does not mean that we let the students avoid the rigor of specification. In fact, at the end of the project the students have to turn the informal models into consistent and complete artifacts. We think that the transition between these modeling techniques needs to be taught in software intensive courses, especially in the design of mobile applications.

We also introduced release management as a workflow in our software engineering project courses to take advantage of the creativity of ambitious students while still ensuring a stable demonstration at the customer acceptance test.

Finally, we presented a case study in which we used these techniques in a multi-customer course with 80 students. Until now in most of our courses, we had been pleased to see 1 or 2 releases, and only one of them at the customer acceptance test. We think that the 163 successful releases the students produced in this multi-customer course is a remarkable result.

7. ACKNOWLEDGMENTS

We would like to thank all participants and customers in our courses, particularly the teaching assistants and the members of our chair, who made these courses possible. We would especially like to thank Helma Schneider for her technical administration, Monika Markl for her organizational help, Uta Weber for managing the finances, and Ruth Demmel for filming. Special thanks to Dan Chiorean and Benoit Combemale for their helpful comments and suggestions. Last but not least we would like to thank Genevieve Cerf for her thorough review of the final version of the paper.

8. REFERENCES

[1] Atlassian. Atlassian software suite, 2012. `http://www.atlassian.com/software`.

[2] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[3] Bit Stadium GmbH. Hockeyapp, 2012. `http://www.hockeyapp.net`.

[4] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[5] B. Bruegge, J. Blythe, J. Jackson, and J. Shufelt. Object-oriented system modeling with OMT. In *OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 359–376. ACM, 1992.

[6] B. Bruegge, J. Cheng, and M. Shaw. A software engineering project course with a real client. Technical Report CMU/SEI-91-EM-4, Carnegie Mellon University, Software Engineering Institute, 1991.

[7] B. Bruegge and A. H. Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java (Third Edition)*. Prentice Hall International, 2009.

[8] B. Bruegge, H. Stangl, and M. Reiss. An experiment in teaching innovation in software engineering: video presentation. In *OOPSLA Companion '08*, pages 807–810, 2008.

[9] B. Bruegge, H. Stangl, and M. Reiss. Dolli 2: project presentation. In *OOPSLA Companion*, pages 1041–1042. ACM, 2009.

[10] J. M. Carroll, editor. *Scenario-based design: envisioning work and technology in system development*. John Wiley and Sons, 1995.

[11] Chair for Applied Software Engineering (TUM). JAMES - Java Architecture for Mobile Extended Services, 1997. `http://www1.in.tum.de/globalse-james`.

[12] Chair for Applied Software Engineering (TUM). iPraktikum, 2012. `http://www1.in.tum.de/static/lehrstuhl_1/projects/451-ios-praktikum-2012`.

[13] E. Dijkstra. Remarks in Panel - Software Engineering: As It Should Be, 1979. Proceedings of the 4th ICSE, Munich, Germany, September 1979.

[14] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[15] W. Maalej, H.-J. Happel, and A. Rashid. When users become collaborators: towards continuous and context-aware user input. In *OOPSLA '09*, pages 981–990, 2009.

[16] D. Mayhew. *The Usability Engineering Lifecycle: A Practitioner's Guide to User Interface Design*. Morgan Kaufmann Publishers, 1999.

[17] J. Nielsen and J. Hackos. *Usability engineering*. Academic press San Diego, 1993.

[18] D. Norman and S. Draper. *User centered system design; new perspectives on human-computer interaction*. L. Erlbaum Associates Inc., 1986.

[19] J. Rudd, K. Stern, and S. Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, 1996.

[20] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Prentice Hall PTR, 2002.

[21] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard business review*, 64(1):137–146, 1986.

[22] J. Tomayko. Teaching a project-intensive introduction to software engineering. Technical Report CMU/SEI-87-TR-20, DTIC Document, 1987.

[23] Visual Paradigm. UML CASE tool for software development, 2012. `http://www.visual-paradigm.com/product/vpuml`.