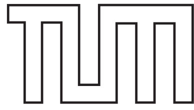
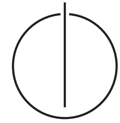
INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN





Forschungs- und Lehreinheit I
Angewandte Softwaretechnik

Rugby - A Process Model for Continuous Software Engineering

Stephan Tobias Krusche

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation:	1.	Univ.-Prof. Bernd Brügge, Ph.D.
	2.	Prof. Dr. Jürgen Börstler, Blekinge Institute of Technology, Karlskrona, Schweden

Die Dissertation wurde am 28.01.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.02.2016 angenommen.

Abstract

Software is developed in increasingly dynamic environments. Organizations need the capability to deal with uncertainty and to react to unexpected changes in requirements and technologies. Agile methods already improve the flexibility towards changes and with the emergence of continuous delivery, regular feedback loops have become possible. The abilities to maintain high code quality through reviews, to regularly release software, and to collect and prioritize user feedback, are necessary for continuous software engineering. However, there exists no uniform process model that handles the increasing number of reviews, releases and feedback reports.

In this dissertation, we describe Rugby, a process model for continuous software engineering that is based on a meta model, which treats development activities as parallel workflows and which allows tailoring, customization and extension. Rugby includes a change model and treats changes as events that activate workflows. It integrates review management, release management, and feedback management as workflows. As a consequence, Rugby handles the increasing number of reviews, releases and feedback and at the same time decreases their size and effort. Rugby reduces the time between development and usage of software features in development projects. When used in education, Rugby reduces the time between teaching and exercising concepts to improve knowledge retention. We applied Rugby in three case studies: (1) in 62 university capstone projects with about 500 students, (2) in a lecture with 57 team projects and about 400 students, and (3) in 8 industry projects with 31 professionals in a company.

Empirical evaluations demonstrate that Rugby significantly increases the frequency and quality of interactions between developers and users as well as instructors and students. (1) The introduction of Rugby increased the number of students in capstone projects who have improved their skills in configuration and release management from 40 % to 80 %. Rugby led to 96 code reviews, 64 releases and 27 feedback reports on average per team in 2014. (2) 77 % of the lecture students who participated in the exercises are confident to apply continuous software engineering workflows in future projects. (3) The 8 industry projects tailored Rugby to their needs and were able to reduce the time effort for integration and delivery from hours to minutes, while increasing the frequency of releases.

Zusammenfassung

Software wird unter immer dynamischeren Bedingungen entwickelt. Organisationen benötigen die Fähigkeit mit Ungewissheit umzugehen und auf unerwartete Änderungen von Anforderungen und Technologien zu reagieren. Agile Methoden verbessern bereits die Flexibilität bezüglich Änderungen und mit dem Aufkommen von kontinuierlicher Auslieferung sind regelmäßige Feedback Schleifen möglich geworden. Die Fähigkeiten hohe Quellcode Qualität durch Reviews zu erhalten, Software regelmäßig auszuliefern und Nutzer Feedback zu sammeln und zu priorisieren, sind nötig für kontinuierliche Software Entwicklung. Jedoch existiert noch kein einheitliches Prozessmodell, das die steigende Zahl von Reviews, Releases und Feedback Berichten behandelt.

In dieser Dissertation beschreiben wir Rugby, ein Prozessmodell zur kontinuierlichen Software Entwicklung, das auf einem Meta Modell basiert, das Entwicklungsaktivitäten als parallele Arbeitsabläufe behandelt und das Tailoring, Anpassung und Erweiterung erlaubt. Rugby beinhaltet ein Änderungsmodell und behandelt Änderungen als Ereignisse, die Abläufe aktivieren. Es integriert Review Verwaltung, Release Verwaltung und Feedback Verwaltung als Arbeitsabläufe. Dadurch behandelt Rugby die steigende Anzahl von Reviews, Releases und Feedback und reduziert gleichzeitig deren Größe und Aufwand. Rugby verringert die Zeit zwischen Entwicklung und Nutzung von Software Bestandteilen in Entwicklungsprojekten. In der Lehre reduziert Rugby die Zeit zwischen Unterrichtung und Einübung von Konzepten und verbessert damit den Wissenserhalt. Rugby wurde in drei Fallstudien angewandt: (1) in 62 Universitätsprojekten mit ca. 500 Studenten; (2) in einer Vorlesung mit 57 Team Projekten und ca. 400 Studenten; sowie (3) in 8 Industrieprojekten mit 31 Fachkräften im Unternehmen.

Empirische Evaluationen zeigen, dass Rugby die Häufigkeit und die Qualität der Interaktionen zwischen Entwicklern und Nutzern sowie zwischen Unterrichtenden und Studenten signifikant erhöht. (1) Die Einführung von Rugby erhöhte die Anzahl der Studenten in Projektkursen, die ihre Fähigkeiten in der Konfigurations- und Release Verwaltung verbesserten, von 40 % auf 80 %. Rugby führte 2014 zu 96 Quellcode Reviews, 64 Releases und 27 Feedback Berichten im Durchschnitt pro Team. (2) 77 % der Studenten in der Vorlesung, die an den Übungen teilnahmen, sind zuversichtlich die Arbeitsabläufe der kontinuierlichen Software Entwicklung in künftigen Projekten anzuwenden. (3) Die 8 Industrieprojekte passten Rugby für ihre Bedürfnisse an und waren in der Lage den Zeitaufwand für Integration und Auslieferung von Stunden auf Minuten zu verringern, während sie die Häufigkeit der Releases erhöhten.

Acknowledgements

Many people influenced the last five years of my life and my research. I want to thank them and acknowledge their support. First, I would like to express my deep gratitude to Bernd Bruegge, who always inspired me with new ideas, with endless enthusiasm and support throughout the whole time of my dissertation. He created an environment of opportunities and growth, supported my ideas, trusted me and my work, provided valuable feedback, encouraged new ways of thinking and always gave me chances to grow and the freedom to try out new approaches.

Further I want to thank Jürgen Börstler for accompanying my dissertation research and for his feedback. I am very grateful to all members of the Chair for Applied Software Engineering. I learned a lot from all of you and I am thankful for all discussions, feedback, fun, and encouragement. In particular, I like to thank my office neighbor Martin Wagner who started with me back in 2011 to organize the capstone course iOS Praktikum, which is one of the foundations of this dissertation. I also want to thank Dr. Yang Li, Sebastian Peters, Constantin Scheuermann and Florian Schneider, who provided valuable feedback.

In addition, I like to thank all coauthors of papers and articles who contributed to this dissertation, in particular Sebastian Peters, Sebastian Klepper, Dora Dzvonyar, and Mjellma Berisha. I would like to express my gratitude to Helma Schneider and Monika Markl, who were always positive and helpful. I am grateful to all study participants for their time, patience, and feedback.

Finally, I want to express my love and gratitude to my family, and in particular to my girlfriend Anke. Writing a dissertation requires even more than the researcher's full attention. I am indebted for your love and devotion, your understanding and your support. Without you, this dissertation would not have been possible!

Contents

Abbreviations **xiii**

1 Introduction **1**
1.1 Existing Process Models in Software Engineering 1
1.2 Problems in Existing Process Models 5
1.3 Motivation for a new Process Model 6
1.4 Research Objectives 7
1.5 Contributions 8
1.6 Dissertation Structure 9

2 Foundations **11**
2.1 Process Models 12
2.2 Version Control 14
2.3 Continuous Integration 17
2.4 Continuous Delivery 18
2.5 Informal Reviews 21
2.6 User Feedback 23
2.7 Learning Techniques 26

3 Rugby’s Process Meta Model **29**
3.1 Static View of Rugby’s Process Meta Model 31
3.2 Rugby’s Change Meta Model 33
3.3 Dynamic View of Rugby’s Process Meta Model 34
3.4 Instantiation of Waterfall Model as Linear Model 38
3.5 Instantiation of Unified Process as Iterative Model 40
3.6 Instantiation of Scrum as Agile Model 42
3.7 Related Process Meta Models 45

4 Rugby's Ecosystem 47
4.1 Top Level Design . . . 47
4.2 Requirements . . . 48
4.3 Use Case Model . . . 56
4.4 Static View of Rugby's Process Model . . . 59
4.5 Dynamic View of Rugby's Process Model . . . 61
4.6 Related Process Models . . . 65

5 Rugby's Workflows 67
5.1 Review Management Workflow . . . 69
5.2 Related Work in the Area of Code Reviews . . . 76
5.3 Release Management Workflow . . . 78
5.4 Related Work in the Area of Release Management . . . 81
5.5 Feedback Management Workflow . . . 84
5.6 Related Work in the Area of User Feedback . . . 88

6 Case Studies 90
6.1 Capstone Course . . . 91
6.1.1 Interventions . . . 92
6.1.2 Course Environment . . . 94
6.1.3 Teaching Approach . . . 100
6.2 Lecture . . . 105
6.2.1 Individual Exercises . . . 107
6.2.2 Team based Exercises . . . 109
6.3 Industry . . . 112
6.3.1 Applicability in Industrial Projects . . . 112
6.3.2 Extending and Customizing Rugby . . . 113

7 Evaluation 117
7.1 Hypotheses . . . 117
7.2 Study Design . . . 119
7.3 Findings . . . 127
7.3.1 Review . . . 127
7.3.2 Release . . . 130
7.3.3 Feedback . . . 131
7.3.4 Frequency . . . 133
7.3.5 Understanding . . . 136
7.3.6 Learning . . . 137

7.3.7 Scalability 141
7.4 Limitations 147
7.5 Summary 152

8 Conclusion 153
8.1 Contributions 153
8.2 Future Work 155

A Terminology 156

B Process Models 160
B.1 Scrum 160
B.2 Unified Process 163

C Rugby's Full Change Model 166

List of Figures 168

List of Tables 173

Bibliography 174

Abbreviations

API .... Application Programming Interface, page 115

AUP .... Agile Unified Process, page 65

BPDM .... Business Process Definition Meta Model, page 10

CASE .... Computer Aided Software Engineering, page 155

CAT .... Client Acceptance Test, page 95

CD .... Continuous Delivery, page 18

CI .... Continuous Integration, page 17

CVS .... Concurrent Version System, page 15

DAD .... Disciplined Agile Delivery, page 65

DoD .... Department of Defense, page 2

FR .... Functional Requirement, page 48

IEEE .... Institute of Electrical and Electronics Engineers, page 22

ITIL .... IT Infrastructure Library, page 83

MOF .... Meta Object Facility, page 29

MOOC .... Massive Open Online Course, page 155

NFR .... Nonfunctional Requirement, page 55

OMG .... Object Management Group, page 10

OOPSLA .... Object Oriented Programming, Systems, Languages and Applications, page 4

POM .... Project Organization and Management, page 105

REST .... Representational State Transfer, page 133

SPEM .... Software Process Engineering Meta Model, page 10

SPMP .... Software Project Management Plan, page 111

SS .... Summer Semester, page 91

SVN .... Subversion, page 15

UML .... Unified Modeling Language, page 30

VCS .... Version Control System, page 14

WS .... Winter Semester, page 91

XP .... Extreme Programming, page 3

Chapter 1

Introduction

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
>
> — Principle behind the Agile Manifesto

The term "software engineering" was first proposed in the 1960s as a reaction to the software crisis in computer industry when software systems became larger and more complex to develop [Mah90]. Projects failed because development concepts and methods were missing and teams worked together in rather chaotic and unstructured ways: the term software engineering "was deliberately chosen as being provocative, in implying the need for software manufacture to be [based] on the types of theoretical foundations and practical disciplines[,] that are traditional in the established branches of engineering" [NRB76, Mah90].

Software engineering was established as a field in 1968 at the NATO conference in Garmisch [NR69]. The vision was to move from unstructured ways to a defined software life cycle model to understand and characterize how software is developed. By adapting successful methods of other engineering domains, e.g. the manufacturing domain, the goal was to industrialize the creation of software with a defined and disciplined process that is well understood, predictable, repeatable and efficient [Fow01].

1.1 Existing Process Models in Software Engineering

Detailed process models emerged with a strong emphasis on planning and the goal to describe the process of engineering software, e.g. Royce's linear waterfall process model [Roy70]. His idea was to transfer the sequential order of activities from production lines in factories to form a linear software development approach. This resulted in

a defined software process following strict rules and avoiding deviations, which were seen as errors that need to be corrected. The hypothesis was that a well defined set of inputs always generates the same output as in production lines.

Linear process models describe an approach that flows steadily through several development activities such as analysis, design, implementation, integration, testing and delivery. The United States Department of Defense (DoD) made the use of the waterfall model mandatory for their software projects [Dep88]. Wong commented on this in 1984: “This [waterfall] model was adopted [...] because [...] software development was guided by DoD standards and these DoD standards impose clean separations between phases. In reality, software development is a complex, continuous, iterative, and repetitive process. The [waterfall] model [...] does not reflect this complexity” [Won84].

The V-Model [JT79, BD93], an extension to the waterfall model, was developed as national standard for the German Federal Armed Forces and has been a mandatory methodology for German governmental projects since 1996 [Bun92]. It emphasizes the separation between development and testing. One advantage in contrast to the waterfall model is that each development phase can be associated with its corresponding test phase and that the model describes the levels of abstractions from coarsest grain to most detailed. In 2004, the V-Model was replaced by its successor, the V-Model XT, which added the capability of tailoring to the initial V-Model [BR05].

One of the drawbacks of linear approaches is the difficulty to accommodate changes after the project has been started. Only after one development activity has been completely finished, the next one can start. Especially in large projects, it is difficult to analyze all requirements completely, correctly and unambiguously, which often leads to a situation, where developers spend an extraordinary amount of time in the analysis phase. This scenario has been well described with the analysis paralysis antipattern [BMMM98]. Another disadvantage is that software is only delivered after it was completely realized at the end of the project so that developers are not able to incorporate external feedback during the development phases.

As a response to this situation, iterative and incremental software process models were developed, such as the spiral model by Boehm in 1988 [Boe88] or the Unified Process by Jacobson, Booch and Rumbough in 1998 [JBR98]. The spiral model includes ideas from rapid prototyping [DMSW98]: initially small but growing increments of the software are constructed that are potentially thrown away in favor of alternative solutions. Each iteration includes a development activity as described in the waterfall model and an additional risk analysis activity to evaluate project continuation multiple times throughout the project. While the spiral model allowed to accommodate changes, the software was still only released after it was completely realized as in linear approaches.

The Unified Process is a use case driven, architecture centric and risk focused process framework. It is decomposed into four different phases: inception, elaboration, construction and transition. Each phase focuses on specific deliverables and can further be split into multiple iterations. The project team addresses critical risks early in the inception and elaboration phases, e.g. by creating an executable architecture baseline, a partial implementation of the system including its core components. Business modeling, requirements, analysis & design, implementation, testing, and deployment are modeled as six core workflows in the Unified Process. A workflow is a thread of related and usually sequential activities performed by the development team in order to produce artifacts. Workflows span across the four phases, while each phase having a different emphasis on specific workflows. In addition to the six core workflows, there are three supporting workflows: project management, environment, and configuration & change management. [JBR98]

Software developers increasingly recognized that the essence of software engineering is to deal with changes and that linear and iterative process models are not capable of addressing this need. Software engineering consists of experimental knowledge work where creativity is important [Bas96]. Such work includes unexpected events, incidents and uncertainty. Lehman realized in the 1980s that software evolution, the continual change to a software system, is inevitably required to keep software up to date with changing environments and to satisfy stakeholders [LB85].

In the 1990's, agile methods emerged with the philosophy that software engineering should not follow a defined process model but rather an empirical model that is structured, but not entirely planned. An empirical process model sees deviations, errors and failures as opportunities that need to be investigated and that lead to adaptions. In 2001, the ideas of empirical process control led to the agile software development manifesto, which values individuals and interactions more than processes and tools, working software more than comprehensive documentation, customer collaboration more than contract negotiation, and responding to change more than following a plan [BBVB+01].

Kent Beck proposed Extreme Programming (XP) as one of the first agile methods, in 1996 in a large Chrysler project [BA04]. XP focuses on development practices such as pair programming and test driven development. Automated tests are written before the actual implementation, and executed on a dedicated integration computer. XP includes the practice of continuous[1] integration where developers integrate source code every few hours, whenever possible, into a source code repository so that integration

[1] Continuous in this context refers to regular practices that are conducted multiple times per day. It does not refer to the mathematical definition of continuity.

failures are detected and repaired early. XP requires a colocated customer, who is always available to the development team and who writes requirements in form of small user stories. It is based on the premise that analysis and design activities should be minimized in the beginning: design emerges from small iterations as the system is developed and opportunities for reuse are identified. [BA04]

Ken Schwaber introduced the agile process model Scrum in 1995 at a workshop at OOPSLA [Sch95], based on a "holistic" method originally described by Tekeuchi and Nonaka in the Harvard Business Review [TN86]. Schwaber and Beedle further described Scrum in 2002 [SB02]. Today, Scrum is used by many software development companies[2]. It describes management aspects and divides development into time boxed iterations with a fixed duration called *Sprint*s, which are usually four weeks long.

In each sprint, the cross functional and self organizing team performs development activities, such as analysis, design, implementation and testing, in parallel to turn defined sprint requirements into a potentially shippable product increment. Within one sprint, no change to these sprint requirements is allowed, however new requirements can be added to the project and existing requirements can be changed or removed, influencing the outcome of future sprints. The development team meets on a daily basis to discuss status, impediments and promises forming a second smaller iteration. This refines the idea of risk analysis and reduces the granularity of risk identification and assessment to just one day.

Anderson formulated the Kanban method for software development as an incremental and evolutionary software process model for organizations in 2007 [And10]. Kanban is another attempt to introduce approaches from engineering into software development. Anderson took ideas from Toyota manufacturing processes implemented in 1953 that use a scheduling system for lean and just in time production [SKCU77]. Kanban focuses on continual improvements and workflow visualizations. In contrast to Scrum, Kanban does not define iterations. Instead it implements a continuous development model with a limitation of parallel work in progress. By limiting the work in progress, the team is able to produce value, minimize waste and reach a consistent development flow [Rei09].

[2] In a survey in 2015 with 3,925 completed responses from a broad range of industries in the global development community, 56 % reported to use Scrum as their choice of agile methodology [Ver15].

1.2 Problems in Existing Process Models

In linear process models, there is a large delay between the development and the usage of the system. Iterative and agile models decreased this delay. With the incorporation of agile methods, software projects have gained more flexibility towards requirements and technology changes, while allowing the early identification of risks through regular meetings. Conboy and Fitzgerald define agility as "the continual readiness of an entity to rapidly or inherently, proactively or reactively, embrace change, through high quality, simplistic, economical components and relationships with its environment" [CF04].

While Scrum embraces change, a limitation is that it does not allow to react to changes within a sprint. It is necessary to specify sprint requirements completely and unambiguously at the beginning of the sprint, which might lead to a smaller form of analysis paralysis. If developers find a problem during the implementation of a requirement, they would need to wait until the end of the sprint, postponing the answer to their problem to the next iteration or they would develop the software in the wrong way. However, changes are uncertain and unpredictable: it is possible, that no changes occur for four weeks and then three changes occur on the same day. Postponing the reaction to unexpected events would increase risks and would lead to inefficient development.

The first two principles in the agile manifesto include customer satisfaction through early and continuous delivery of valuable software in short cycles and the welcome of changing requirements, even late in the development phase [BBVB+01]. Jez Humble defined a deployment model for continuous delivery in 2010 [HF10], that is based on continuous integration: the development team keeps the software artifacts in a state so that the software and changes to it can be released at any time. Regular releases lead to an improved product quality and high customer satisfaction [Che15].

However, Humble's model does not describe how to handle changes and user feedback. In fact, Rodríguez and her colleagues state in their systematic mapping study in 2016 that "a clear research gap exists for mechanisms to use customer feedback in the most appropriate way so that information can be quickly interpreted" [RHL+16]. Humble's model also does not cover an approach to prevent poor internal software quality with respect to architecture, design and source code, such as quality assurance through software reviews.

Jan Bosch and his colleagues introduced the term continuous software engineering [Bos14] and the stairway to heaven model to describe the organizational transition from traditional development over continuous deployment to research and development as

innovation system [OAB12]. While feedback is covered in this model, it is limited to the output of user monitoring. Quality assurance through reviews is not mentioned.

There has not yet been an attempt to model software engineering continuously with reviews, releases and feedback as central concepts. This is the focus of this dissertation.

1.3 Motivation for a new Process Model

Based on Takeuchi's and Nonaka's article [TN86], we use the term Rugby[3] for a process model for continuous software engineering that can be used in university and industry. Rugby is an attempt to improve the quality throughout the process by allowing changes and reaction to these changes anytime in the development process. Instead of waiting until the end of the project in linear approaches or until the end of an iteration in iterative and agile approaches, Rugby allows to address unexpected events immediately. Development teams can detect changes earlier by validating the usage of realized requirements with users. Rugby provides a mechanism to release software event based, i.e. when developers want to obtain feedback.

In Rugby, changes can occur at any time in the development process and are seen as events waking up processes and triggering reactions[4]. When there is a new requirement, the analysis process wakes up. A new technology wakes up the design process and a new crash triggers the implementation and testing processes to wake up. Sometimes a bug turns out to be a problem in the requirements. For this reason, Rugby starts with the assumption, that all activities can react to changes at any time. The development activities have to work together and cannot be viewed isolated.

Rugby is activity based and provides a general view on how to react to unexpected changes. It describes continuous workflows for review management, release management and feedback management and their relationship with the development team. Rugby is tailorable on the process level and customizable on the workflow level to specific project environments with respect to the development activities. Its change model includes many predefined events and is extensible for new events so that it can also be applied in other areas, such as eduction.

[3]In the sport of Rugby, a scrum is a method to restart play after a foul or when the ball has gone out of play. In that sense, Scrum handles only the exceptions of the game. The rule that a Rugby player passes the ball laterally to another player running in parallel formation is a better metaphor for describing the continuous interaction between developers, customers and users.

[4]This idea is based on the metaphor of communicating sequential processes by Hoare [Hoa78].

The high frequency of conducting reviews, releasing software and obtaining user feedback reduces the difficulty of these activities [Fow11]. Rugby combines multiple feedback loops from its review, release and feedback workflows as shown in Figure 1.1. There are multiple steps (review, integration, test, delivery) between the development and use of software which can lead to failures (e.g. *Integration Failure*) producing change events. If feedback is needed immediately, developers can bypass most steps and deliver unreviewed, unintegrated and untested code (only the step delivery has to pass). Rugby treats user feedback also as change event.

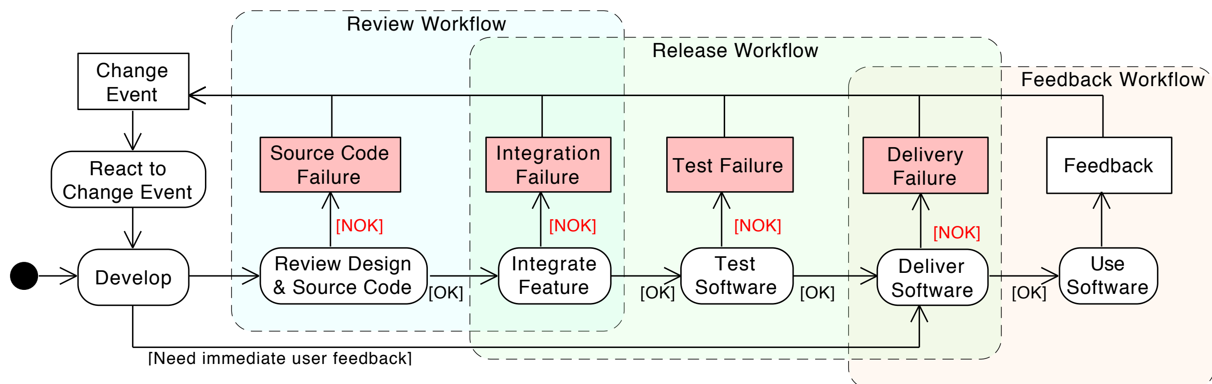


Figure 1.1: Interaction between review, integration, test, delivery and feedback loops in Rugby

Rugby has been designed to be extensible to other domains. It can e.g. also be used in education where it provides an environment that reduces the time between the delivery of a concept and the corresponding exercise to deepen the knowledge about the concept. As a result, Rugby allows the decomposition of large lectures into multiple smaller units. By combining theory and exercises in small units shortly after each other, it is easier for students to understand **and** to apply the learned concepts.

1.4 Research Objectives

The main research objective pursued in this dissertation is to reduce the delay between development and usage of software in development projects and to reduce the delay between teaching a concept and exercising it. Our hypothesis is that Rugby is able to reduce these delays through an increased frequency of delivery and feedback, which reduces the size of the content – software changes in the development and concepts in education – and which increases the amount of interaction. Rugby's process model integrates three workflows that address this hypothesis:

(1) The continuous review management workflow improves the internal quality because multiple persons review design decisions and code changes before they are

integrated into the main codebase. It increases the understanding about code and design within the team through peer reviews and ensures that only high quality is present in released product increments.

(2) The continuous release management workflow automates integration, testing, and delivery to improve productivity and efficiency, and increases the product quality of the software, because it reduces the amount of bugs. It allows the creation of event based releases as product increments during an iteration to clarify questions, to obtain user feedback or to review the current progress.

(3) The continuous feedback management workflow allows developers to build the right product with the right features. It improves the motivation of users to give feedback by simplifying the feedback provision. It decreases the effort for developers to analyze, prioritize and integrate user feedback with a semi automatic approach. Developers can choose to integrate changes directly or to postpone them to future iterations.

Another objective is to create a process meta model that allows the instantiation of multiple process models by generalizing the concepts using an event based change model. We describe the static and dynamic aspects of the meta model and show three examples of instantiations for a linear, an iterative and an agile process model. Rugby's process model is also an instantiation of the process meta model.

1.5 Contributions

The main contribution of this dissertation is the establishment of a software process model for continuous software engineering that is based on the agile principles of Scrum [SB02] and the iterative workflows of the Unified Process [Kru04]. Rugby models software engineering as a set of continuously running processes which are waiting for events. Based on the Unified Process, we call these independent processes "workflows". Each of these workflows models a software engineering activity such as requirements elicitation, analysis, design, implementation and testing. Rugby in particular includes workflows for review management to achieve high design and code quality, for release management to achieve automated integration and delivery, and for feedback management to allow the assessment and integration of user feedback.

We demonstrate Rugby's applicability in three case studies in university and industry. In a first case study, we applied Rugby's workflows in 62 university capstone course projects between 2011 and 2015. We conducted multiple formative evaluations to analyze the introduction of Rugby's workflows in the capstone courses. Qualitative studies allowed us to understand the perspectives and motivations of development teams and

showed that Rugby improves the communication between developers and users. Using quantitative analysis, we measured the usage of the activities of Rugby's workflows in the capstone projects. From 2011 to 2014, the number of students in capstone projects who have improved their skills in configuration and release management increased from 40 % to 80 %. Rugby led to 96 code reviews, 64 delivered releases, 136 downloads and 27 feedback reports on average per team in 2014.

Another contribution is the application of Rugby in a large lecture environment based on experiential and blended learning. In a second case study, we used Rugby in a university course in 2015 which demonstrates Rugby's extensibility. A qualitative study showed that Rugby improves the communication between instructors and students and the participation of students. We used Rugby in lecture units to decrease the delay between teaching a concept and exercising it and identified a correlation between the exercise participation and the final exam results. A lecture unit consists of the presentation of a concept, followed by an exercise that ensures the students apply the concept immediately after the presentation and a retrospective. 77 % of the students who participated in the exercises are confident to apply continuous software engineering workflows in future projects. We identified a correlation between the exercise participation and the final exam results.

In a third case study, we applied Rugby in eight industry projects in 2014, which demonstrated Rugby's customizability for different project environments. The industry projects tailored Rugby and customized its release management workflows to their needs. A qualitative analysis confirmed the results that we found in the capstone projects in university and demonstrated that the delay between development and usage is reduced. The professionals in the eight industry projects were able to reduce the time effort for integration and delivery from hours to minutes, while increasing the frequency of releases.

1.6 Dissertation Structure

The dissertation is organized as follows[5].

Chapter 2 describes the foundations of the dissertation and introduces the terminology. First, we discuss the differences between defined and empirical process models. Then we describe existing concepts in version control, continuous integration, continuous delivery, informal reviews and user feedback. Finally, we present existing learning techniques that we apply in education in the first and second case study.

[5] Instead of a separate chapter for related work, we relate our work to others in different sections after the description of corresponding concepts.

Chapter 3 describes the static and dynamic aspects of Rugby's process meta model. It consists of a change model where changes are treated as events that activate certain workflows. We show that the meta model can be instantiated with linear, iterative and agile software process models. In addition, we relate Rugby to the Software Process Engineering Meta Model (SPEM) [Obj08b] and the Business Process Definition Meta Model (BPDM) [Obj08a], which were both standardized by the Object Management Group (OMG).

Chapter 4 presents Rugby's ecosystem with its environments and Rugby's process model. It describes Rugby's requirements and use case model and shows Rugby's static and dynamic aspects. We relate Rugby to the Disciplined Agile Delivery [AL12] by Scott Ambler, which is the successor of the Agile Unified Process [CPP10], a hybrid process model that combines the Unified Process with agile aspects.

Chapter 5 describes the three workflows review management, release management and feedback management which are integrated into Rugby's process model. It also describes related work in the area of code reviews, release management and user feedback.

Chapter 6 presents the application of Rugby in three case studies. The first case study describes a capstone course where several interventions improved the learning experience of students. In the second case study, we use Rugby's workflows to improve the learning experience of students in the context of a large lecture based on blended learning principles. The third case study demonstrates the applicability and customizability of Rugby in industrial projects in a company.

Chapter 7 describes empirical evaluations that we conducted in the three case studies. We describe the design of six qualitative studies using questionnaires and personal interviews as evaluation methods and three quantitative studies. We show the findings of these studies and describe limitations of the evaluations.

Chapter 8 concludes the dissertation by summarizing its contributions and by providing different ideas for future work.

Appendix A lists the terminology and the used definitions. **Appendix B** describes more details about the process models Scrum and Unified Process because Rugby is based on ideas of both. **Appendix C** shows Rugby's full change model.

The dissertation is based on previously published journal articles as well as conference and workshop papers: [BKW12], [KA14], [KABW14], [KB14], [DKA14], [KKP+15], [BKA15], [KBB16], [DKAB16], [KKB16].

Chapter 2

Foundations

“We all need people who will give us feedback. That‘s how we improve.“

— Bill Gates

This chapter introduces the foundations for this dissertation. Section 2.1 discusses the differences of defined and empirical process control and introduces terms related to process models. It also describes the concepts of process tailoring and customization. In Section 2.2, we describe version control concepts, discuss the differences between central and distributed version control and describe a model for branch and merge management.

Section 2.3 discusses the concept of continuous integration where developers integrate their software frequently, i.e. several times per day. In Section 2.4, we describe the idea of continuous delivery that builds on top of continuous integration and allows to deliver changes to the software easily, frequently and automatically throughout the whole software lifecycle. Distributed version control, continuous integration and continuous delivery are the components of Rugby’s release management workflow.

Section 2.5 discusses the differences between formal and informal reviews and introduces a taxonomy of tool assisted code reviews. Distributed version control including branch and merge management and code reviews are the components of Rugby’s review management workflow. In Section 2.6, we discuss user involvement and user feedback as important techniques for software evolution, and present a taxonomy of typical roles in the user feedback process. User feedback is a component of Rugby’s feedback management workflow. Section 2.7 describes different learning techniques such as blended and experiential learning that we use in the application of Rugby in education. Appendix A summarizes the terminology introduced in this chapter.

2.1 Process Models

The term process is used in various contexts and can be defined as “a related set of activities conducted to the specific purpose of product definition” [Rol93]. A process model describes how activities must, should or could be done in contrast to the process itself which is what really happens. A process model corresponds to the way of working prescribed by the methodology in use. A process is an instantiation of a process model and includes workflows describing work practices in the project. A workflow is a thread of cohesive and mostly sequential activities performed by project participants that produce artifacts [BD09]. A process corresponds to a concrete project whose output is a product such as the implemented information system. The knowledge required to design a process model is related to process meta modeling [Rol93].

Process control deals with mechanisms for maintaining the output of a process in a specified and desired range. Control does not mean the process can be completely predicted. One goal of process control is the minimization of risks. If potential problems can be detected early, a reaction can be defined to solve the problem. An example for process control is a heating system with a feedback loop. High temperature in the environment reduces heating, low temperature increases heating.

There are different styles of process models distinguished by their process control, in particular deterministic vs. nondeterministic process control. Deterministic process control requires every element of the process to be completely understood. Given a well defined set of inputs, a defined process will generate the same outputs and results until completion every time. A defined process is a collection of tightly coupled steps: the output of one step is the input to the next step [SB02].

A defined process model is an example for deterministic process control. It has no random variables. The defined control model handles changes and failures as deviations that need to be corrected and tries to prevent the occurrence of changes in the plan or outcome of the process. An example for defined process control is European navigation in the open sea as described by Gladman [Gla64]. The European navigator follows a specific route. If he deviates from the route in his voyage, he introduces corrective activities to achieve the planned goal and to return to the planned route. This type of process control is used in assembly productions as described by Taylor in the 1920s [Tay14] for the production of similar industry products such as cars.

However, it cannot be used for building complex systems which require creative problem solving and adaptivity to change [BD09]. Software development is a complex process with random variables, that cannot be defined completely deterministic. “It is typical to adopt the defined (theoretical) modeling approach when the underlying

mechanisms by which a process operates are reasonably well understood. When the process is too complicated for the defined approach, the empirical approach is the appropriate choice" [OR94]. Complex processes, which are not understood completely, require an empirical control model, which is an example for a nondeterministic process control.

Empirical process control includes visibility, inspection and adaption. It allows to control complex processes, which cannot perfectly be defined and which would generate unrepeatable and unpredictable results. Visibility means that aspects of the process that affect the outcome must be known and visible. Inspection requires that process aspects are evaluated frequently so that unacceptable variances can be detected. The process should be adapted if one or more aspects are in an unacceptable range.

The empirical model described by Schwaber is based on Ogunnaike's definition of a stochastic model [OR94]. It handles changes and failures as opportunities. A quick reaction to changes can lead to advantages compared to competitors. If random variables are allowed, the process control is nondeterministic and empirical. If no random variables exist, the process control is deterministic and defined.

An example of an empirical process control is the Polynesian navigation as described by Gladwin [Gla64] and Suchman [Suc07], which it is goal oriented and uses heuristics. Polynesian navigation begins with an objective instead of a plan. It responds to conditions when they arise using skills and experience by observing the environment such as wind, waves, tides, stars, etc., to figure out the direction towards the objective or the proximity of land. A management style like the Polynesian navigation is the key of agile methods such as Scrum [Sch95]. It is particularly useful in projects exploring new technologies whose purpose is to break existing paradigms [BD09].

Consequently, a realistic model of the reality would employ probability distributions to characterize random process variables. The process model then becomes stochastic, with individual outcomes driven by random variables drawn from the given probability distributions [Kel91]. Stochastic control is not solved analytically and deals with the existence of uncertainty. It includes one or more stochastic elements, i.e. random components which cannot be completely predicted: it is unknown if and when certain events happen.

An agile manager uses empirical process control to expect the unexpected, but has confidence that the chaos can be organized. He has contingency plans to handle unexpected situations such as impediments and changes in technologies or requirements. The random variable is not the unexpected situation itself, it is the point in time, when the situation occurs. If something unexpected occurs, the process is inspected and potentially adapted. There are three ways to adapt a process model to the domain

and environment of a project: process tailoring, process customization and process extension.

We define process tailoring as adapting a process to operational needs on a higher level through removing, modifying or adding specific workflows, without significantly deviating from the process model. According to Ginsberg, two terms are used to describe a relationship, e.g. identification, correlation, and/or derivation, between different levels of process definitions: *interpreting* and *tailoring*. Interpreting is the act of analyzing and correlating the definition of a general process description with respect to an existing instantiation. The goal is to understand the relationship between the description - the process model - and the instantiation - the process. Tailoring is the act of adjusting the definition of a general process description - the process model - to derive an alternative environment - the process. [GQ95]

We define process customization as adapting a process to operational needs on a smaller level through removing, modifying or adding specific activities to an existing workflow, without significantly deviating from the workflow model of the process. In the sense of inheritance in object oriented systems, process customization can be seen as adding a new sub class that can modify the behavior of an existing super class, e.g. by adding new static or dynamic aspects and by overriding methods.

We define process extension as adapting a process to operational needs through adding a new workflow or activity that cannot be described with existing elements of the process model. In the sense of inheritance in object oriented systems, process extension can be seen as adding a new super class with completely new static and dynamic aspects that cannot be described with existing classes.

2.2 Version Control

A central concept in software configuration management [Mor04] is version control. A version control system (VCS) is "a system that records changes to a file or set of files over time so that you can recall specific versions later" [Cha09]. From the early days of software engineering, developers sought a way to store and organize the different versions of files that emerge during software development. They started to copy and move files from time to time to a local backup directory.

Initially, the management of these files was tedious and error prone, when it was still performed manually. Moreover, accumulating the entire history of a certain file was a complex and time consuming task. As a first improvement, developers began using local version control systems, which are essentially local databases, to keep track of

the files and their changes. This solved the very basic problems of local backup directories. In relation to architectural patterns, local version control systems correspond to a standalone or monolithic architecture, since they are completely independent of other systems [Cha09].

With growing demands for more complex software, team collaboration became the norm. At this point, a local database for file management was not sufficient anymore. When a developer begins editing an already existing file, he does not wish to work on his latest local version, but instead wants to get the latest changes from the team. Similarly, when viewing a file's change history, a programmer is not only interested in his own changes, but also in what the other team members have accomplished. To meet these demands, the former local database was moved to a server which was then shared by the entire team.

This marked the introduction of centralized version control systems, the most popular of which include: Concurrent Version System (CVS), Subversion (SVN) and Perforce. To modify a file, the programmer would check out the file to his local machine, the client, and depending on the configuration of the version control system, he would need to block it from his team members. Centralized version control systems use the repository architectural pattern, a special case of the client server pattern [Sha96, CGB+02]. A central property of the repository pattern is the existence of many loosely coupled clients that mainly interact with a database on a server, but very little with each other [Cha09].

Centralized version control systems introduced a significant weakness: the servers became single points of failure. A single server crash could destroy the whole team's documents and work history. This problem is solved by the currently most used type of version control systems: distributed version control systems. In modern distributed systems like git or Mercurial, clients mirror the full repository, making them very robust to machine failures. Distributed version control systems use an implementation of the peer to peer architectural pattern. Since programmers can create branches and commit locally, they also allow for very fast and comfortable offline development [Cha09, CGB+02].

The basic entity of distributed version control is the repository, which is a place where files can be stored along with their history. Developers do not work directly on the files from the repository, but instead have a local copy of the repository, called the working copy, from a specific point in time. The process of initially creating this working copy is the check out. After a developer has checked out a local copy from the repository, he can start developing by making changes to the files of the local copy. When he is done, he has the option of committing his changes to the repository. If the

corresponding files in the repository have not been changed in the meantime, he can simply proceed with the commit. Otherwise, if the changed files have simultaneously been modified by commits from other developers, a conflict occurs. Before the developer can commit his changes, he needs to resolve the conflict by either manually or automatically merging the conflicted files.

If the developer is aware of the fact that he will work on a specific set of files at a speed different than that of his colleagues, he can opt for creating a ”parallel version of the repository” [Cha09], namely a branch. This enables the existence of multiple copies of the files in the repository, so that the developers can each work on them independently [Cha09]. While centralized version control systems such as Subversion are easier to use and faster to learn, distributed version control systems such as git provide more possibilities, in particular to commit locally (offline) and to create and merge branches fast and easily [BCSD14]. Easier branching allows context switches and exploratory coding [MBNC14]. A branching workflow defines the coexistence of versions, i.e. when new branches are created, merged and deleted. Branch management is the activity of defining and controlling these workflows [Ins88, WS02]. Branching models handle different types of branches, e.g. feature, bugfix, release and hotfix branches. An established branching model is git flow [Dri10] which is shown in Figure 2.1.

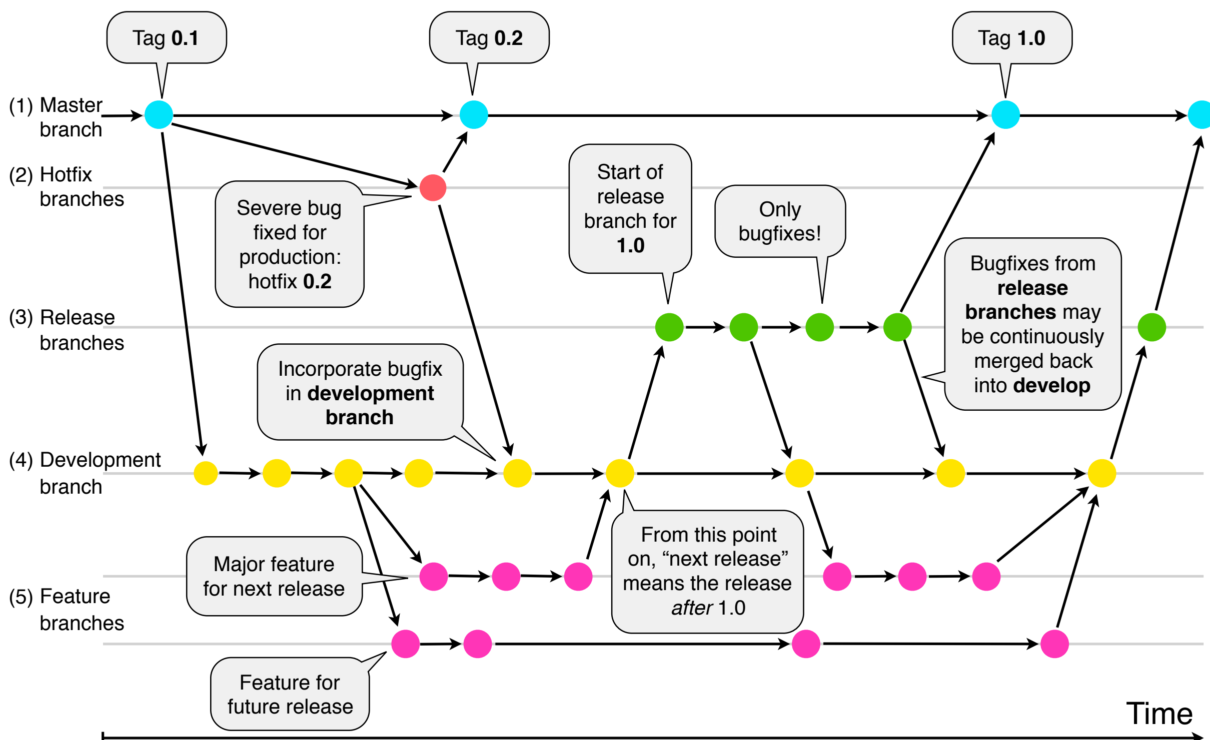


Figure 2.1: Git flow branching model (adapted from [Dri10])

Git flow defines five branch types: (1) A single master branch is used to store versions in form of tags that were released to production. (2) Hotfix branches are used to fix bugs in production leading to minor version updates. They only live very shortly in case of critical bugs. (3) Release branches prepare new major releases. They are created when all features for the new release are finished and additional time for testing and bug fixing is needed. If the release is finished, the changes on the release branch are merged to the master branch and the release branch is closed. Additional bugfixes happen in hotfix branches. (4) A single development branch is used to integrate finished features, hotfixes and bugfixes from release branches for future releases. (5) The realization of requirements happens on feature branches. Multiple feature branches can run in parallel. For each requirement, a feature branch is created and merged back to the development branch as soon as the feature is finished.

2.3 Continuous Integration

Continuous integration (CI) is a practice first described by Grady Booch [Boo91] as “a far better way to measure productivity” and to avoid a big bang integration at the end of the project. Software is integrated regularly during development, multiple times per day: “No code sits unintegrated for more than a couple of hours. At the end of every development episode, the code is integrated with the latest release and all the tests must run at 100 %” [BA04]. Kent Beck included continuous integration as practice in Extreme Programming [BA04] emphasizing the importance of face to face communication over technological support.

Martin Fowler defines continuous integration as “software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including regression testing) to detect integration errors as quickly as possible” [Fow06]. The term “continuous” is not used in its mathematical definition and does not imply that the integration is constantly running and never stops. It is rather a description of a continual, regular use, usually multiple times per day, but at least once a day. Small and frequent changes are immediately verified by an integration and test system, and in case of a broken integration, they are immediately fixed. This practice can highly reduce the time between a defect is introduced and when it is fixed [DMG07].

CI reduces integration effort and makes integration work more predictable, compared to performing integration after a large amount of development work. The goal

of CI is to have a working state of the project throughout the development. When practicing continuous integration, the development team uses an automated workflow to periodically integrate and test all parts of the system and to continuously verify the correct functionality of a system after making changes. When a test fails, team members turn their attention to find the reason and to fix the problem behind the test failure. Continuous integration leads to the early detection and removal of errors, improved release frequency and predictability, increased developer productivity, and improved communication [SB14].

2.4 Continuous Delivery

Continuous delivery (CD) builds on top of continuous integration. In addition to integration, it automates all steps in the delivery process so that even small changes to software, such as bug fixes, can be released efficiently without much manual effort. CD provides rapid feedback to developers, improves software quality and reduces integration and delivery risks. Martin Fowler recommends to provide an easy way to obtain the latest version and to automate the deployment process [Fow06]. The idea is to frequently test the software in real use and simultaneously make the process of delivering the software to its target environment fast and error resistant. Humble and Farley expand on this and describe continuous delivery as the practice of automating the entire process of taking a change from commit to release [HF10].

The aim of continuous delivery is to achieve "a reliable, predictable, visible, and largely automated process with well understood, quantifiable risks" by combining continuous integration with other advanced development practices such as software configuration management, data management, environment management, and release management [HF10]. Chen defines continuous delivery as a "a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time" [Che15].

Continuous delivery builds on the goal of having an integrated and working version at all times and extends it to being able to release the latest version at all times. In contrast to the related concept of continuous deployment, the goal is not to actually deploy every version to the target environment, but the ability to do that [HF10]. The transformation of continuous delivery should not be limited to the software development team, but should also consider other functions, in particular sales and marketing [NS13]. This suggests that an end to end consideration of the software development lifecycle is important.

The ability to create releases for each change in the source code in a fast, easy and robust way is the purpose of continuous delivery. The release manager is responsible for a release, has the final authority which changes are included in the release and usually triggers the delivery [Ere03]. Humble models a deployment pipeline as a stage gate process that is shown in Figure 2.2. During its lifecycle a build moves from the integration stage through multiple test stages to a delivery stage which uploads the binaries to the target environment. In each stage, the build is checked against certain quality criteria. If these are fulfilled, the build is promoted to the next stage and can be delivered to a target environment, e.g. production, with no effort.

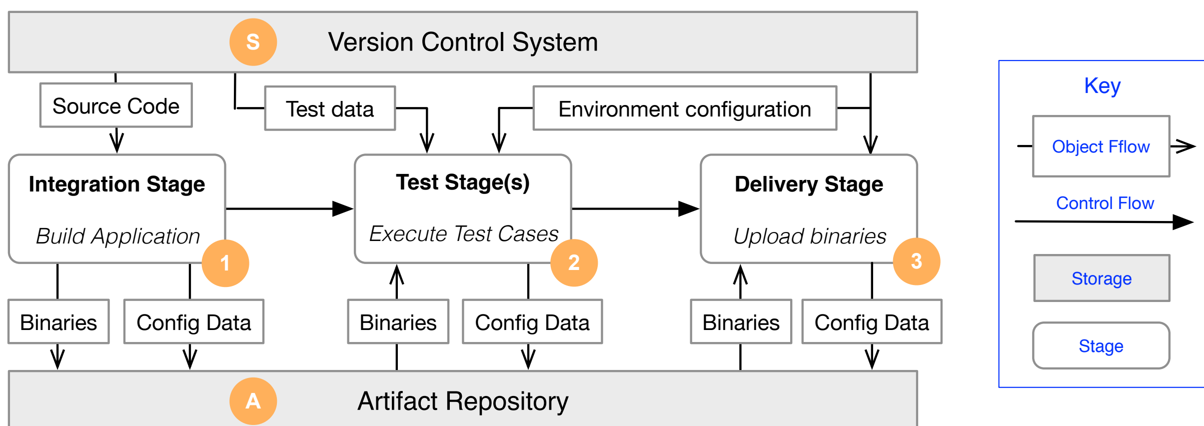


Figure 2.2: Deployment process with integration stage, test stage(s) and delivery stage (adapted from [HF10])

Continuous delivery can provide the following benefits [Che15, LMP+15]:

- **Accelerated time to market:** business value inherent in new releases comes to customers more quickly
- **Building the right product:** frequent releases let the development team obtain user feedback more quickly
- **Improved productivity and efficiency:** time and effort savings through automation
- **Reduced risk of a release failure:** the release process becomes more reliable
- **Improved product quality:** the number of open bugs and production incidents decrease
- **Improved customer satisfaction:** developer can respond to feedback more quickly to increase the level of satisfaction
- **Improved collaboration:** Closer connection between development and operations

In addition, continuous delivery may pose the following challenges [Che15, LMP+15]:

- **Different interests:** varying interests in different departments of an organization could lead to conflicts
- **Heterogeneous organizations:** different development and production environments might require different workflows
- **Resistance against change:** traditional and bureaucratic processes may hinder continuous delivery
- **Long test execution for large software:** large test suites might require a long test execution time
- **Manual and nonfunctional testing:** Some tests can hardly be automated and require manual testing

On an organizational level, the “stovepipe” or “silo” antipattern may hinder an efficient implementation of continuous delivery when the structure of the organization restricts the flow of information, inhibiting or preventing cross organizational communication [Han93]. One example of such an antipattern is a stovepipe between the development and the operations department. A concept related to continuous delivery is DevOps (**Dev**elopment + **Op**eration**s**), a term that describes improvements in the collaboration between development and operations departments by automating processes and focusing on important issues [Hum11].

DevOps is used to describe a collection of “practices that advocate the collaboration between software developers and IT departments with the goal to shorten the feedback loop and align the goals of both” [CSA15]. A continuous delivery infrastructure can be used to implement DevOps, i.e. to deliver a new software release to customers, to collect user feedback and usage data, and transfer it back to developers. Activities of the development and operations departments are combined to prevent stovepipes and to improve release engineering.

DevOps and release engineering both try to improve product value for the customer by enabling the organization to react to changes faster and to deliver high quality software [DPL15]. Dyck and his colleagues define DevOps as “organizational approach that stresses empathy and cross-functional collaboration within and between teams – especially development and IT operations – in software development organizations, in order to operate resilient systems and accelerate delivery of changes” [DPL15]. They define release engineering as “a software engineering discipline concerned with the development, implementation, and improvement of processes to deploy high-quality software reliably and predictably” [DPL15].

2.5 Informal Reviews

Reviews habe the purpose to increase quality. Product quality has been investigated from various perspectives. Renown quality experts either take the stance that quality means ”conformance to requirements” [Cro80] or define it relative to the user’s needs and their ”stated or unstated, conscious or merely sensed” [Fei02] requirements. Another component of quality includes patterns, which describe generic solutions for recurring problems within a particular context using proven concepts [GHJV94]. The application of the pattern has consequences in addition to the benefits. When change occurs and the consequences become “decidedly negative” [BMMM98], patterns devolve into antipatterns. An antipattern has a refactored solution: a “commonly occurring method in which the antipattern can be resolved and reengineered into a more beneficial form” [BMMM98]. Another knowledge base for recognizing mistakes are code smells, defined as “indicators that usually correspond to a deeper problem in the system” [Fow99].

We define quality as conformance to flexible specifications that respond to the changes of the user’s needs, in addition to the usage of corresponding patterns to address nonfunctional requirements if applicable, while avoiding antipatterns. We consider code quality to be a subclass of quality, focusing on functional requirements, system architecture, design patterns and coding guidelines, avoiding development antipatterns and code smells. Refactoring becomes essential for improving quality, as it helps to remove both code smells and problematic solutions from antipatterns. The Oxford Dictionary defines review as “formal assessment of something with the intention of instituting change if necessary” [Ste10]. This definition applies to software engineering, where reviews are an important quality assurance method to check for defects, deviations from development standards, and other problems in products [CLB03].

Weinberg states how early software developers, even the likes of von Neumann and Babbage, understood that correctness was too difficult a task to master by oneself, and sought their colleagues’ feedback [WF84]. These initial reviews were informal in nature, as there was no defined or agreed upon process. In fact, well defined reviews eluded research interests into the 1970s. Weinberg [WF84] explains that “the need for reviewing was so obvious to the best programmers that they rarely mentioned it in print, while the worst programmers believed they were so good that their work did not need reviewing”. An early approach to reviews found in literature is a formal and well defined process called inspection [Pat05]. Software inspections were developed under the direction of Michael Fagan in an effort to improve quality and increase productivity. Fagan’s inspection process consists of six activities: planning, overview, preparation,

inspection, rework and follow up. The first three lay out the foundation: the author determines what materials are to be inspected and ensures that they meet predefined entry criteria. Then, a meeting is scheduled, the participants are chosen and each is assigned one of the four inspection roles: designer, coder, tester and moderator. This set of roles ensures that the material is reviewed from various perspectives to identify different bugs. [Fag76, Fag86]

Walkthroughs are lower in formality than inspections [Pat05]. There are different approaches to define the process ranging from formal ones such as Yourdon's structured walkthrough [You79] or the IEEE Standard 1028 [Ins08], to informal ones that focus mainly on the walkthrough meeting [SLS14, Blu92]. They all have in common the fact that the author walks an audience of reviewers step by step through the material, while explaining the purpose and reasoning behind it. There is consensus on the purpose for walkthroughs to evaluate and improve the quality of the materials by finding defects, suggesting (alternative) solutions, checking conformance to standards, educating the audience on the materials and training new team members.

Informal reviews provide a higher flexibility in contrast to formal ones, to be conducted as needed and to have the ability to customize the process. Moreover, planning is limited to choosing the reviewers and asking them for feedback. They often only include the code author and a few reviewers with programming or testing background [Pat05]. The review results do not need to be explicitly documented. Listing the remarks or revising the document is in most cases sufficient. Thus, informal reviews can be simplified down to a cross reading in an author reader cycle [SLS14]. Informal reviews have mainly focused on source code, due to the tools developed to help con-

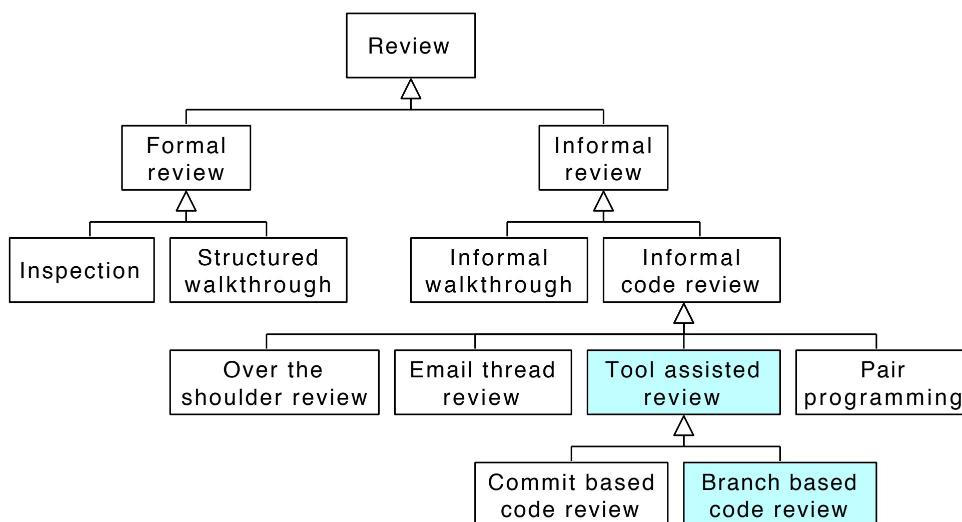


Figure 2.3: Review taxonomy (adapted from [CBDT06])

duct code reviews and the techniques specific to programming. Figure 2.3 shows a taxonomy for reviews including variations for informal code reviews.

A code review is a “manual assessment of source code by humans, mainly intended to identify defects and quality problems” [BBZJ14]. Beller et al. and Bacchelli and Bird define the term modern code review, which “is characterized by fewer formal requirements, a tendency to include tool support, and a strive to make reviews more efficient and less time-consuming” [BBZJ14, BB13]. A code review should not only focus on the source code itself, but also take into account the architecture and the object design of the particular software. An additional activity specific to code reviews is the integration needed to merge the individual code changes with the project’s shared codebase which resides in a repository under version control (see Section 2.2). The rest of the review process is tailored to apply to source code. The informal code review process typically includes the following four activities as shown in Figure 2.4: preparation, examination, rework, and integration. Depending on the review type, activities are skipped to make the process lighter and more flexible.

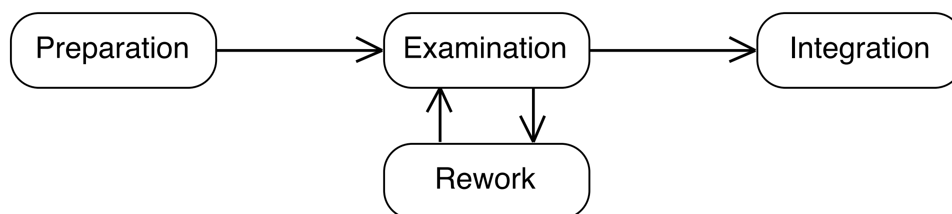


Figure 2.4: Activities in informal code reviews (adapted from [CBDT06])

There are variations of informal code reviews, which Cohen [CBDT06] categorizes as: over the shoulder review, email thread review, tool assisted review and pair programming [CW00]. We distinguish two types of tool assisted reviews: commit based and branch based code reviews. The difference between them is whether the review is conducted on a single commit or on an entire branch that usually contains multiple commits. In this dissertation, we describe an informal review technique for tool assisted reviews that can be instantiated as workflow: branch based code reviews.

2.6 User Feedback

The IKIWISI (“I’ll know it when I see it”) phenomenon [Boe00] describes that users are frequently not able to express their needs and expectations from scratch, but quite good at criticizing existing software. Without having an executable prototype of the software, it is difficult for project stakeholders to discuss important issues [KB14]. User involve-

ment can be defined as “a systematic exchange of information between (prospective) users and developers”, aiming for a better understanding of user needs and a consequent improvement of the software [Pag13].

User involvement is usually mentioned in connection with terms like user input [MHR09], user participation [Cav95], or participatory design [Dam96]. Involving users in the software development process has been recognized as an important source of information for development teams [Hol05], and has shown to have a positive impact on user satisfaction and project success [Kuj03]. While user involvement alone is no guarantee for the success of a project [Cav95], statistical research has shown that it significantly increases the requirements quality [KKLK05].

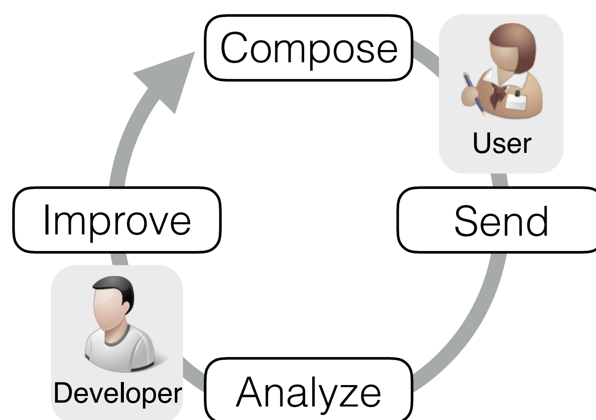


Figure 2.5: Circular model of feedback in software evolution (adapted from [Sch11])

User feedback can be defined as an artifact in the user involvement process, because ”providing feedback is a user interaction [...] to communicate her subjective experience with the application to the application developers” [Pag13]. User feedback contains important information for developers and helps to improve software quality and to identify missing features [PB13]. Software evolution refers to the activity of developing and continuously adapting software due to changes in the environment or needs of its stakeholders, including both development and maintenance activities [Men08]. Figure 2.5 shows the circular model of feedback in software evolution [Sch11]: Users **compose** and **send** feedback to developers who **analyze** the feedback and **improve** the software according to the feedback. In a future release, users evaluate the improvements, compose additional feedback and the cycle continues. User involvement strategies include the user’s opinion at varying stages of the software evolution process.

In the early phases of a project, users can contribute to the idea generation process and describe their needs regarding the software, whereas in the later stages

they can evaluate the implemented functionality and identify possibilities for improvement [Ala02]. Users can both validate existing requirements, e.g. by giving their opinion on a prototype, and also state new requirements which could provide a competitive advantage in the future [KDoD+03]. Users should be involved continuously to guarantee that not only features, but also nonfunctional requirements such as usability and performance are satisfactory [KK05].

The domain of user feedback includes different stakeholders who can take on multiple roles, which we summarize in the taxonomy in Figure 2.6. A stakeholder can fulfill multiple roles, but depending on the role his perspective and thus also his requirements towards a system for user feedback might differ [BVGW10].

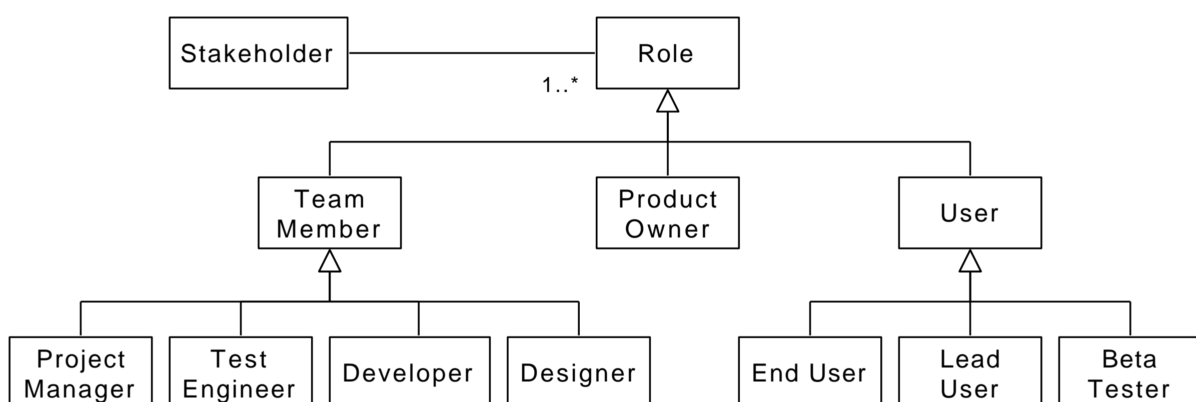


Figure 2.6: Exemplary taxonomy of roles in the user feedback process (non exhaustive, adapted from [DKAB16])

We divide the development team into the roles *Project Manager*, *Test Engineer*, *Developer* and *Designer*, although other roles are possible depending on the organizational structure. These roles have varying needs for user feedback, for instance a designer is more interested in feedback which concerns overall design adjustments than in a bug report, while a test engineer is likely looking for information to reconstruct a problem in the test environment. The role of the *Product Owner* is defined as in Scrum (compare Appendix B.1). His interests in a system for user feedback include the ability to provide feedback and reporting capabilities. For instance, he might check the number of bug reports for a specific release of the application as an indicator for software product quality.

The role of the *User* can be defined as "The people who (will) use the delivered software application" [Roe15]. Several researchers categorize users according to their experience either in software development [YF07, Cav95] or the application domain in question [CMPP08]. The technical background of a user is relevant, since there

is an information gap between what a user provides and what a development team needs [ZPB+10]. Users who have experience in software engineering know which information a development team needs to convert feedback into a requirement, thus they are more likely to include this information in their feedback [Roe15, ZPB+10]. We distinguish three user roles depending on their experience and their motives regarding user feedback: end user, lead user and beta tester.

An *End User* is "any organizational unit or person who has an interaction with the computer based information system as a consumer or producer/consumer of information" [CK89]. A *Lead User* is someone whose present needs will become general in the future and who benefits significantly by having those needs met [Hip86]. Lead users are likely to try out a new application or technology earlier than the majority of the user base, e.g. by downloading a prerelease version of a software, which makes their interests comparable to those of early adopters. In addition to their interest in innovation, lead users are also keen on their needs being realized, which motivates them to provide feedback to the development team of an application [Cav95]. A *Beta Tester* does not necessarily have a technical background, but has the task of testing an application in its target environment, typically before an official release to the whole user base [BD09]. Since beta testers have the task of ensuring the application's quality and finding possible errors, they are most likely to give feedback and to provide a sufficient amount of detail for their findings.

2.7 Learning Techniques

Software engineering is an activity that requires collaboration and practical application of knowledge. Educators struggle when teaching it in traditional lecture based environments where activities take place in the front of the classroom. Lectures are usually similar to broadcasting, where essential interactions are initiated by the teacher with only limited participation on the students side. Self guided learning, personal responsibility, practical relevance and individualization are important elements of a great learning experience. Several pedagogic theories have been developed that include these elements.

Problem based learning is a technique to learn about a subject through the experience of problem solving. Educators facilitate learning by supporting, guiding, and monitoring this process. Working in groups, students identify what they know, what they need to know, and how and where to access new information that leads to the resolution of the problem [BF98].

Cooperative learning is an educational approach which aims to organize classroom activities into social learning experiences: Students work in groups to complete tasks collectively towards a common goal. The teacher's role changes from giving information to facilitating students' learning. Everyone succeeds when the group succeeds [J+91].

Blended learning allows students to learn through delivery of content and instructions via computer mediated activities, digital media and online media. While still attending traditional teaching environments, face-to-face methods are combined with computer mediated activities. Blended learning facilitates a simultaneous, independent and collaborative learning experience [GK04].

Experiential learning is the process of learning from experience, a methodology in which educators engage with students in direct experience to increase knowledge, develop skills, and clarify values. Aristoteles said: "For the things we have to learn before we can do them, we learn by doing them". John Dewey followed this idea with his statement that "there is an intimate and necessary relation between the process of actual experience and education" [Kol84].

While the combination of these learning techniques leads to a more complex experience for educators, it lowers their stress and leads to higher satisfaction [BAKHE03]. A Chinese proverb, first mentioned by Confucius and adapted by Benjamin Franklin describes a modern approach to exercise based education. In recent publications an extended version of the proverb is mentioned:

> „ *Tell me and I will forget.*
> *Show me and I will remember.*
> *Involve me and I will understand.*
> *Step back and I will act.* "

— Chinese proverb [KKLW01]

The first line *"Tell me and I will forget"* describes that explaining a concept only theoretically does not give students the possibility to apply it. The second line *"Show me and I will remember"* includes the idea of cognitive apprenticeship: an apprentice observes the skills of a master who shows how a concept works in practice, e.g. in a tutorial. Clarifying the thinking process behind the application of the concept makes it easier for the apprentice to imitate the behavior [CBH91].

The third line *"Involve me and I will understand"* includes aspects of experiential learning. Involving students in the learning process, e.g. by using interactive tutorials,

allows them to apply a concept on their own, possibly in a different way that fits to their own techniques. It helps them to understand a concept together with its application. The last line *"Step back and I will act"* refers to self guided learning, self improvement and problem based learning. Students take the responsibility to solve a certain problem on their own using the concepts they learned before. This only happens if the educator steps aside and allows students to act on their own. This proverb is the foundation of how we teach software engineering in capstone courses and in lectures [KRTB16] that we present as case studies in Chapter 6.

Chapter 3

Rugby’s Process Meta Model

> “Practical experience has shown the need for modeling software engineering entities (especially processes), measuring those entities, reusing the models, and improving the models.“
>
> — Hans Dieter Rombach and Martin Verlage

In this chapter we introduce Rugby’s process meta model. It abstracts core concepts that can be found in different types of process models: linear, iterative, agile and continuous process models. Figure 3.1 shows Rugby’s process meta model integrated into the meta object facility (MOF) with four layers (from most abstract at the top to most concrete at the bottom) and corresponding examples. The topmost layer M3 consists of the meta meta model with the most abstract concept of a class.

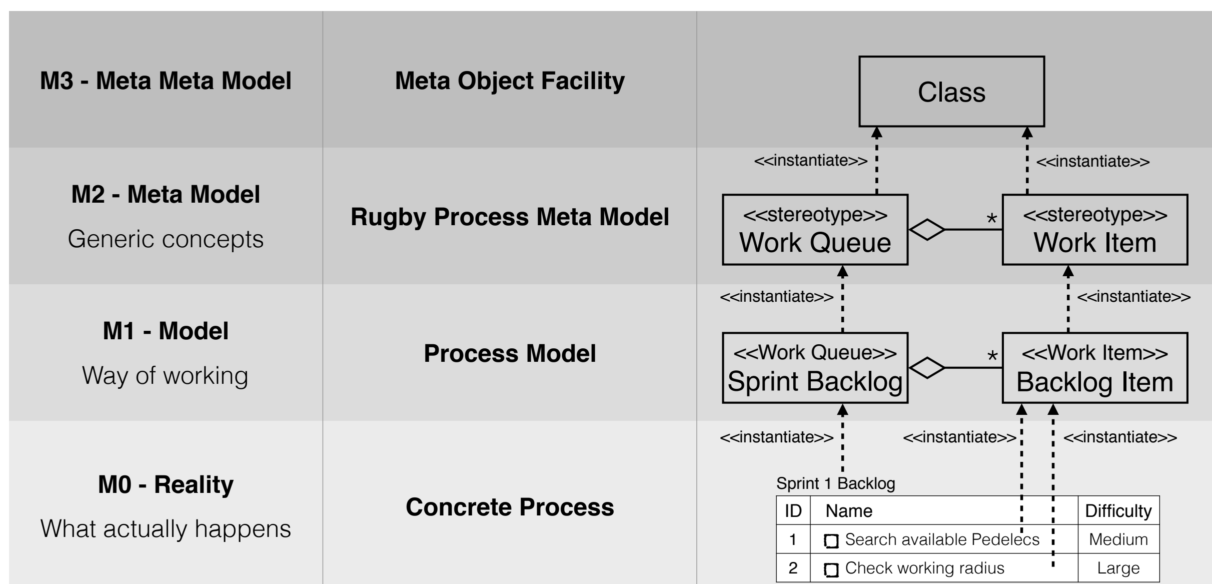


Figure 3.1: Rugby’s process meta model in the meta object facility layers

Rugby's process meta model resides in the M2 layer where generic concepts are modeled that can be found in all process models. An example of a generic concept on M2 is the *Work Queue*, an abstract representation of an ordered list of *Work Item*s with a priority. Section 3.1 shows a static view of Rugby's process meta model including other generic concepts, their attributes and operations, and their relations as UML (Unified Modeling Language) class diagram. Rugby's change model is described in Section 3.2. It models changes as events that trigger the activations of workflows which can subscribe to events and generate events. Section 3.3 shows the dynamic aspects of the meta model including activities and reactions to events using UML activity diagrams and a UML state chart diagram.

On layer M1 in Figure 3.1, there can be different process models that describe the way of working in a software development project. An example of a concrete concept in Figure 3.1 on the M1 layer is the *Sprint Backlog* which is an instantiation of the *Work Queue* on the M2 layer. The sprint backlog consists of *Backlog Item*s in Scrum, which are instantiations of *Work Item*s. Instantiations are represented using the stereotype notation, e.g. <<*Work Queue* >>, which is similar to an inheritance relationship. We show four instantiations of Rugby's meta model, which all reside on the M1 layer. In Section 3.4, we describe the waterfall model [Roy70] as an instantiation of a linear process model. Section 3.5 shows static and dynamic views of the Unified Process [JBR98] as an instantiation of an iterative process model. In Section 3.6, we describe Scrum [Sch95] as an instantiation of an agile process model. The instantiation of Rugby as process model for continuous software engineering is shown in Chapter 4.

Concrete processes used in projects are instances of process models and reside on the M0 layer in Figure 3.1, that describes what actually happens in the reality. An example would be the real *Sprint 1 Backlog* of a project written on a whiteboard or paper. The example in Figure 3.1 includes two concrete backlog items "Search available Pedelecs" and "Check working radius" with specific values for attributes such as ID and difficulty. We describe concrete projects and their elements in the case studies in Chapter 6.

In Section 3.7, we relate Rugby's process meta model to two other process meta models, the Software & Systems Process Engineering Meta Model Specification (SPEM) [Obj08b] and the Business Process Definition Meta Model (BPDM) [Obj08a]. Both meta models were specified by the Object Management Group (OMG).

3.1 Static View of Rugby's Process Meta Model

Rugby's process meta model allows to start with a deterministic process control model (compare Section 2.1) and to change to an empirical process control if required, which makes the meta model tailorable and flexible. If the development process supports change of the requirements during development, a change model can be included. If the development supports multiple releases, a release workflow can be included. If the development process supports user feedback, a feedback workflow can be included. If the development allows multiple reviews during development, the Rugby meta model allows the instantiation of a review management workflow. Rugby's process meta model consists of five packages that are shown in Figure 3.2.

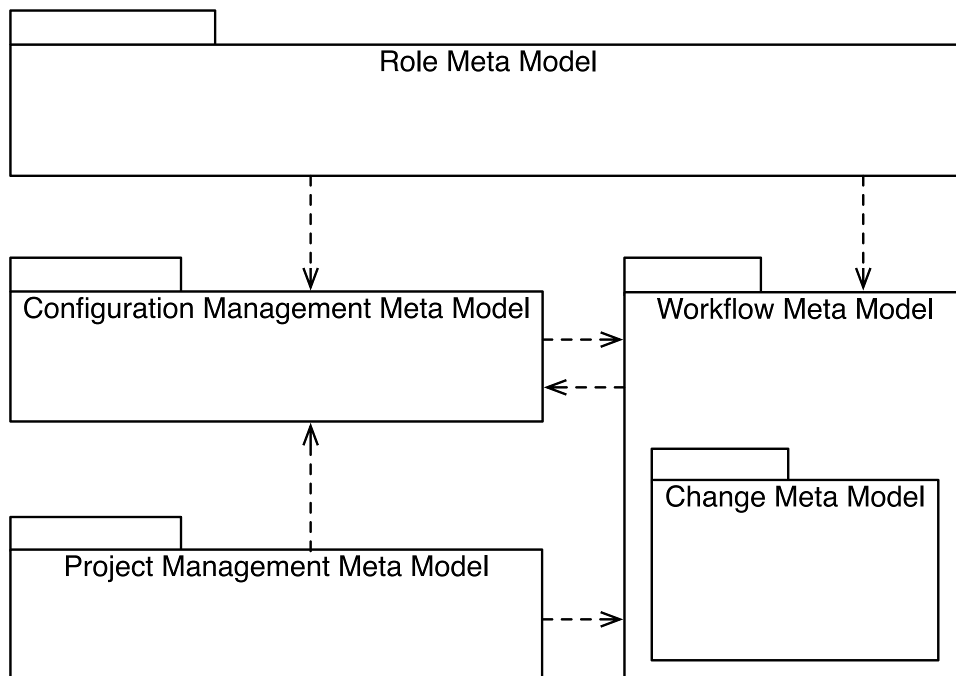


Figure 3.2: Overview of Rugby's process meta model

The role meta model describes generic project roles and how they are integrated into the organization. It depends on the configuration management meta model, which includes controlled items, and on the workflow meta model. The workflow meta model describes how parallel workflows can be customized and activated by events of the change meta model inside of the workflow meta model. Configuration management and workflow meta model depend on each other. There is also a project management meta model that describes other process aspects such as phases and milestones and that depends on the workflow and the configuration management model.

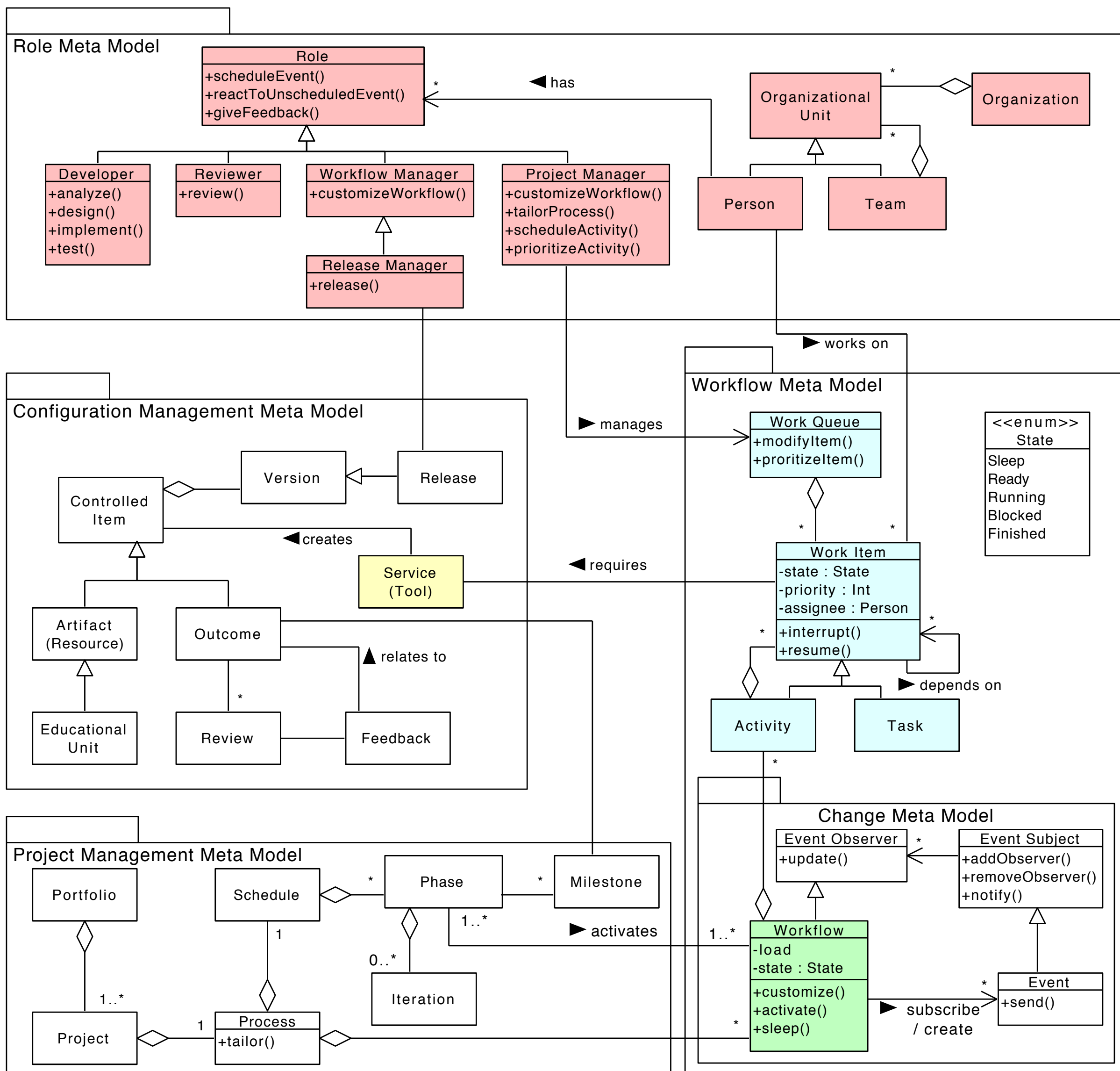


Figure 3.3: Static view of Rugby's process meta model as UML class diagram describing the core concepts and their relationships. The event hierarchy is described in more detail in Figure 3.4.

Figure 3.3 shows the static view of Rugby's process meta model. The role meta model includes a composite pattern for *Organizational Unit*s that build *Team*s of *Team Member*s (and potentially teams of teams). Each team member can have multiple roles such as *Developer* or *Reviewer*. A *Project Manager* is responsible for tailoring the process to the specific needs of a project environment and to customize workflows together with the *Workflow Manager* who is responsible for a specific workflow. For instance, the *Release Manager* is responsible for *Release*s in the release management workflow.

The project manager also manages the *Work Queue* by scheduling and prioritizing an *Activity* which is a subclass of a *Work Item*. A work queue is a prioritized list of work items. New elements are sorted into the queue after their priority and urgency. If priority or urgency of a work item change, the element can be moved in the queue. There might be cases where a team member decides to start with a different work item than the first one in the queue, e.g. due to dependencies or due to capacity reasons.

The workflow meta model includes a composite pattern to decompose activities, which belong to workflows, into smaller work items, such as *Task*s. Work items can depend on each other and can have different states, such as ready, running or finished. The change meta model includes the workflow as an *Event Observer*, which can subscribe to certain *Event*s and also create events that trigger the activation of other workflows.

A workflow belongs to the *Process* of the *Project* in the project management meta model. The process includes a *Schedule* that consists of multiple *Phase*s. There are process models, in which certain workflows are only activated in specific phases. Each phase can have *Milestone*s, e.g. an important meeting where the continuation of the project is decided or where an *Outcome* of the project is discussed, e.g. the documentation of the requirements analysis phase.

Outcomes are *Controlled Item*s and belong to the configuration management meta model. Controlled items are outcomes of work items created by team members using *Service*s (*Tool*s) and have a *Version*. The same document might exist in multiple versions after it was changed due to feedback. Certain outcomes (e.g. builds) are combined into a *Release*, which is a specific version. *Resource*s, such as *Educational Unit*s, e.g. documentation, tutorials or manuals, are also controlled items. An outcome can be subject of a *Review* that leads to *Feedback* on the outcome.

3.2 Rugby's Change Meta Model

Rugby's change meta model includes different *Event* types. In Rugby, an event is the generic form of a change and can trigger interruptions of the current workflow and lead to the activation an another workflow. This is modeled with the observer pattern. Figure 3.4 shows a simplified version with some exemplary events of the event taxonomy with. The full model with all events used in this dissertation is shown in Figure C.1 in Appendix C. Rugby's event model is extensible: Events can be added, changed and removed through process tailoring, workflow customization or the use of

Rugby in different domains such as education. New Rugby events can be added during a project.

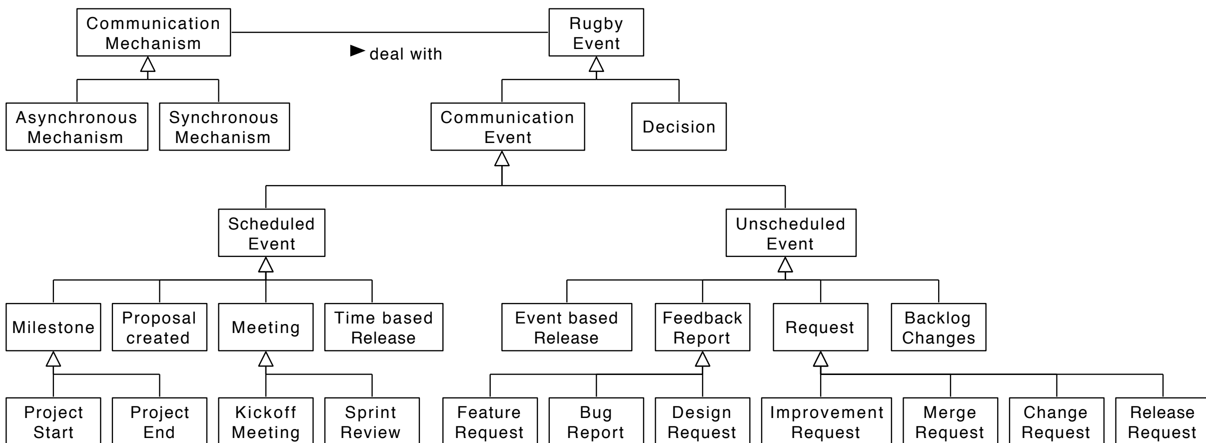


Figure 3.4: Simplified version of Rugby's Change Model including an event taxonomy for scheduled and unscheduled events. The full model can be found in Figure C.1.

Rugby's change model distinguishes between scheduled (e.g. *Meeting*s) and unscheduled (e.g. a *Bug Report* or *Change Request*) events and tries to provide a unified model for both. An event only notifies a workflow if it is interested, i.e. if it subscribed to the event. Workflows create events and notify other events when they are active. For instance, a bug report might be interesting for the implementation workflow, while a feature request might be more interesting for the analysis workflow. An important distinction in Rugby's meta model is made between time based releases, which are scheduled, e.g. at the end of a Sprint in Scrum, and event based releases, which are not scheduled and happen e.g. if developers need feedback or customers request a new release. Other examples of scheduled events are milestones, such as *Project Start* or *Project End*.

3.3 Dynamic View of Rugby's Process Meta Model

Figure 3.5 shows a dynamic view of Rugby's process meta model describing the control flow of a workflow. When the project starts, all workflows are started with the *Workflow Start* event. Workflows subscribe to events they are interested in and then immediately sleep until a *Rugby Event* occurs which they subscribed.

If such an event occurs, the workflow is activated and performs the work until the work is finished. Then, the workflow notifies other workflows about the finished work by sending a *Rugby Event*. A specific event, which each workflow subscribes to, is the *Workflow Customization*. It is created, when a workflow manager, who is responsible

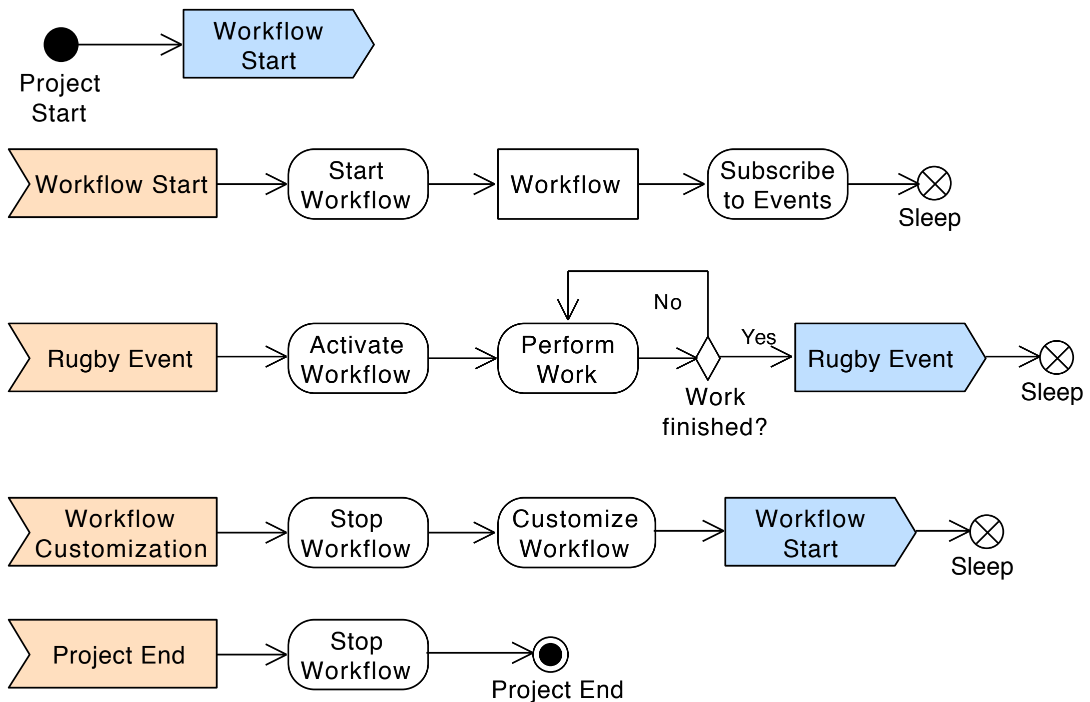


Figure 3.5: Dynamic view of Rugby's process meta model as UML activity diagram describing the core activities and their control flow

for the workflow, wants to customize a workflow. Before the customization, the workflow is stopped. The workflow manager can then customize the workflow and start the workflow again. The workflow might now subscribe to different events. Such workflow customizations must be communicated to the whole team, so that everyone is aware of the customization. At the project end, all workflows are stopped.

Figure 3.6 shows an exemplary instantiation of multiple workflows communicating with each other through Rugby events. An arrow represents a *Rugby Event* that activates another workflow. Multiple workflows run at the same time.

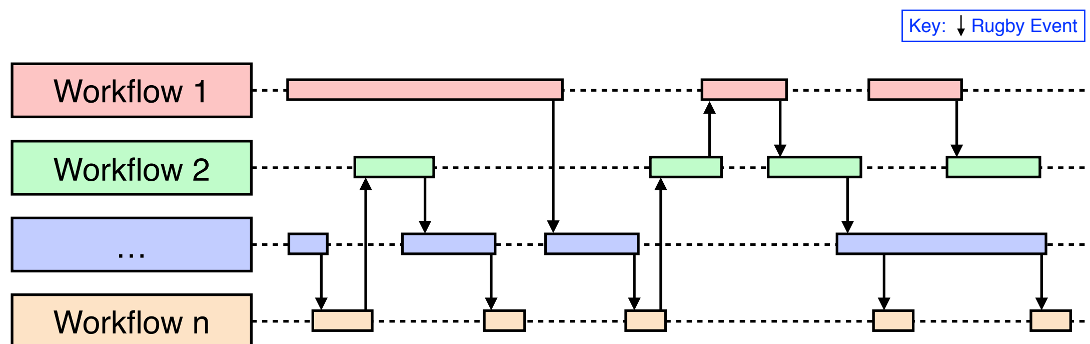


Figure 3.6: Generic example for the communication between multiple workflows through Rugby events

Figure 3.7 shows the lifecycle model for a workflow as UML state machine. Immediately after the workflow has been started and has subscribed to events, it transitions to the *Sleep* state. Rugby events trigger the transition to the *Active* state. Workflows can be interrupted so that they are in the state *Blocked*. This happens e.g. if important information is missing that is necessary to perform the work. If the problem is solved, e.g. the information is available, the workflow can resume and transition to the active state again. If the work is performed, the workflow omits a Rugby event and transitions to the sleep state again. The workflow might also transition to the *Finished* state, when it is stopped, e.g. for customizations or when the project ends.

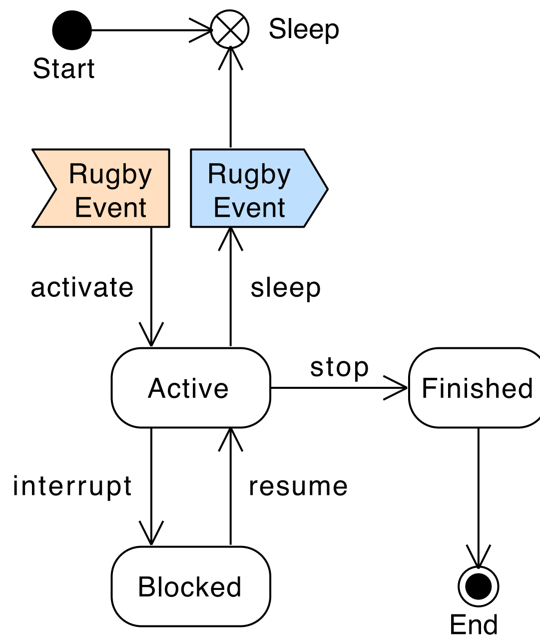


Figure 3.7: The lifecycle model of a Rugby workflow (UML State machine)

Figure 3.8 shows an exemplary instantiation of Rugby's work queue which is based on the concepts of a scheduler with process priorities [Hil92]. On the left side, the queue is visualized as a priority list with different work items, such as a feature request. On the bottom, the priority is low and on the top of the list, the priority is high. Different types of work items can be included in the work queue. The work queue allows changes, e.g. if a user reports an important bug (colored red in Figure 3.8), this bug is inserted into the work queue on a high position and the work for it is started immediately. A design improvement (colored green) however might not be important and is inserted on a low position.

Depending on the type of the work item, a specific sequence of workflows is performed by team members, which is shown on the right side of Figure 3.8. Review, release and feedback workflows are highlighted in grey, as they are core workflows in

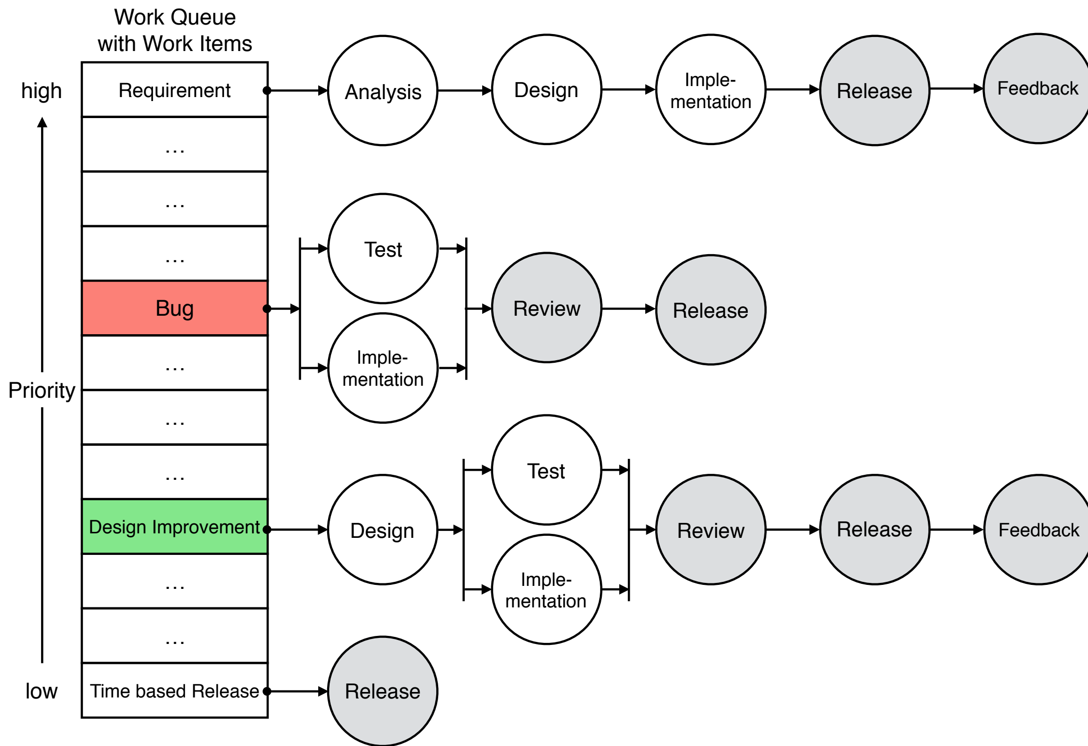


Figure 3.8: Exemplary instantiation of Rugby's work queue (adapted from QNX Microkernel's process priorities [Hil92])

Rugby. A bug can e.g. be implemented and tested immediately, while a feature request first needs to be analyzed and designed before it can be implemented. The creation of a time based release is a work item where only the release workflow needs to run.

Some of these workflows might be performed in parallel for a work item, e.g. implementation and testing. Each workflow type in Rugby's process meta model can be instantiated multiple times for different work items, such the design workflow in Figure 3.8, so each workflow can be part of multiple, parallel threads. The workflow has a load property in percentage, so that it can be balanced. Sleeping workflows have a load of 0, workflows with one team member working full time have a load of 1 (100 %). It is possible that multiple team members work on a workflow so that its load is higher than 1. In general the sum of all workflow loads cannot be higher than the number of team members. This is expressed by the following formula, where w is the workflow, n is the number of all currently running workflows, $w_1, ..., w_n$ are the workflows, l_i is the load of the workflow w_i between 0 and t, and t is the number of team members.

$$\sum_{i=1}^{n} l_i \leq t$$

3.4 Instantiation of Waterfall Model as Linear Model

The first example of an instance of Rugby's process meta model is the linear waterfall model, which was first described by Royce [Roy70] and which uses a defined process model control. Its static view is shown in Figure 3.9.

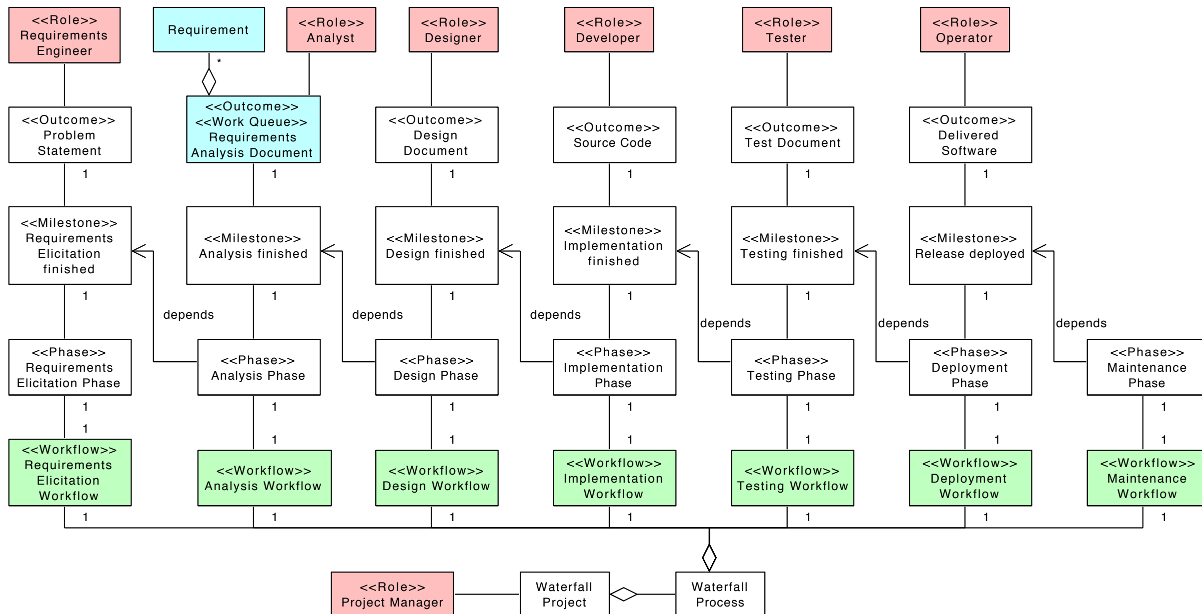


Figure 3.9: Static view of the Waterfall process model with a UML class diagram as an example of a linear instantiation of Rugby's process meta model (adapted from [Roy70])

The model has a one to one mapping between phases and workflows. It includes seven phases that correspond to seven workflows: *Requirements Elicitation*, *Analysis*, *Design*, *Implementation*, *Testing*, *Deployment* and *Maintenance*. All phases produce an outcome and depend on a milestone of the previous phase that represents the end of the phase, e.g. the implementation phase can only start if the design phase is finished and the design document was completely realized. There is only one release produced after the testing phase that is further adapted in the maintenance phase. The *Requirements Analysis Document* consists of multiple prioritized requirements and can be seen as a static instantiation of Rugby's work queue. Once the document is completely realized and the priorities are determined, it should not change anymore in the Waterfall model.

Figure 3.10 shows the dynamic view of six workflows in the waterfall model (excluding maintenance). The *Requirements Elicitation Workflow* starts directly after the *Project Start* event and lasts until the *Problem Statement* is realized which triggers the event *Requirements Elicitation finished* that is observed by the *Analysis Workflow*. Each workflow in the Waterfall model subscribes to only event, which is triggered by

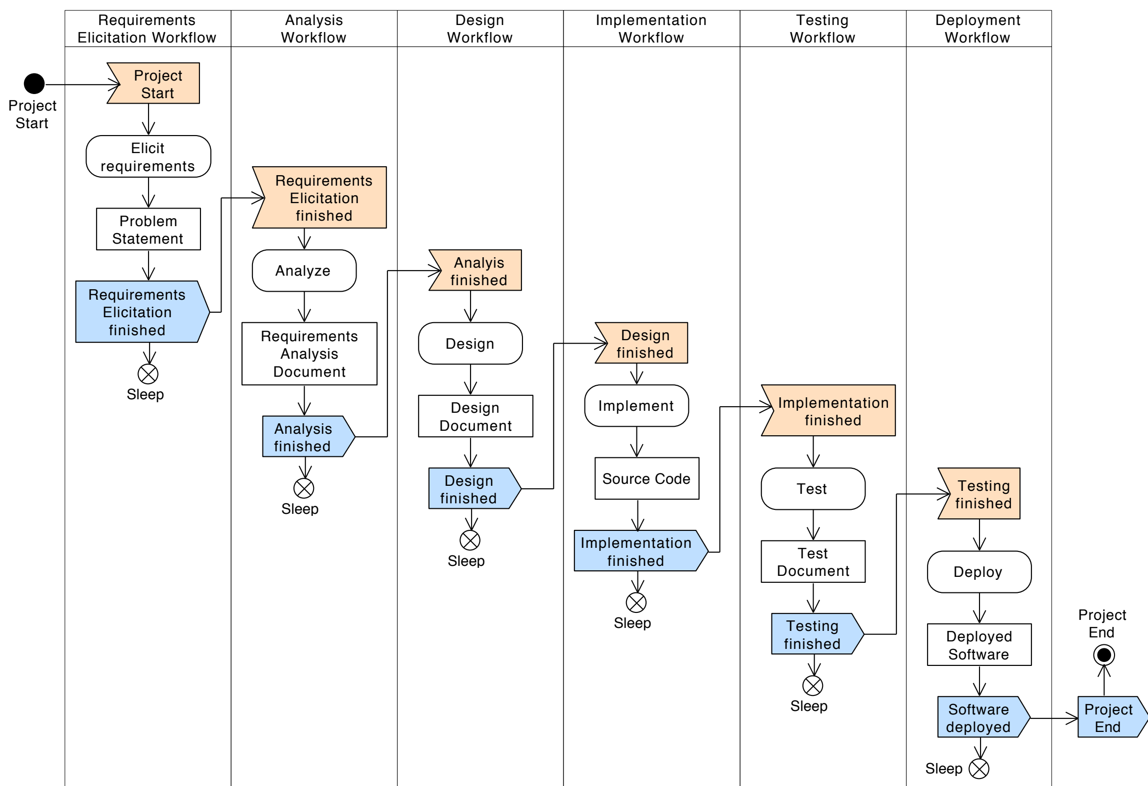


Figure 3.10: Dynamic view[1] of the Waterfall process model with a UML activity diagram as an example of a linear instantiation of Rugby's process meta model (adapted from [Roy70])

the end of the previous phase, and ignores all other events. After activation, a workflow runs solely until its outcome was produced. Then the workflow sleeps again and the next one starts.

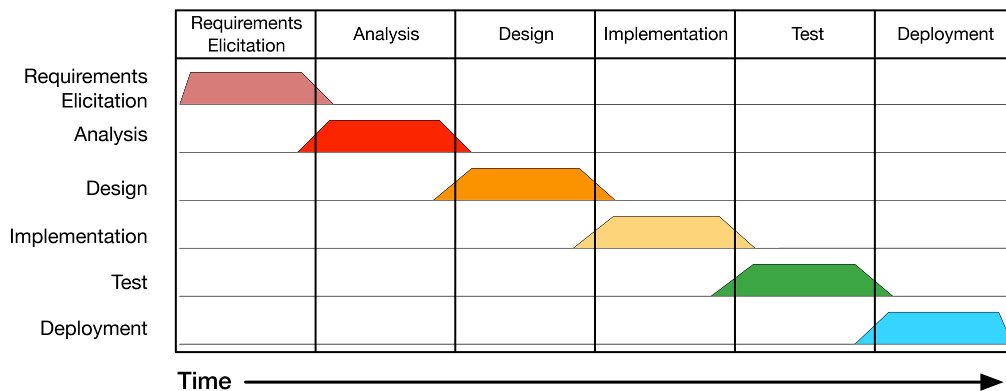


Figure 3.11: Lifecycle of the Waterfall process as view of its dynamic model: Illustration of the relative emphasis of workflows in the Waterfall model (adapted from [Roy70])

[1]This is a simplified view that leaves out the transitions to previous phases of the Waterfall model that are also described in the initial publication by Royce [Roy70]. Even a defined process might have a change control process with a software configuration control board as e.g. defined in [BF14].

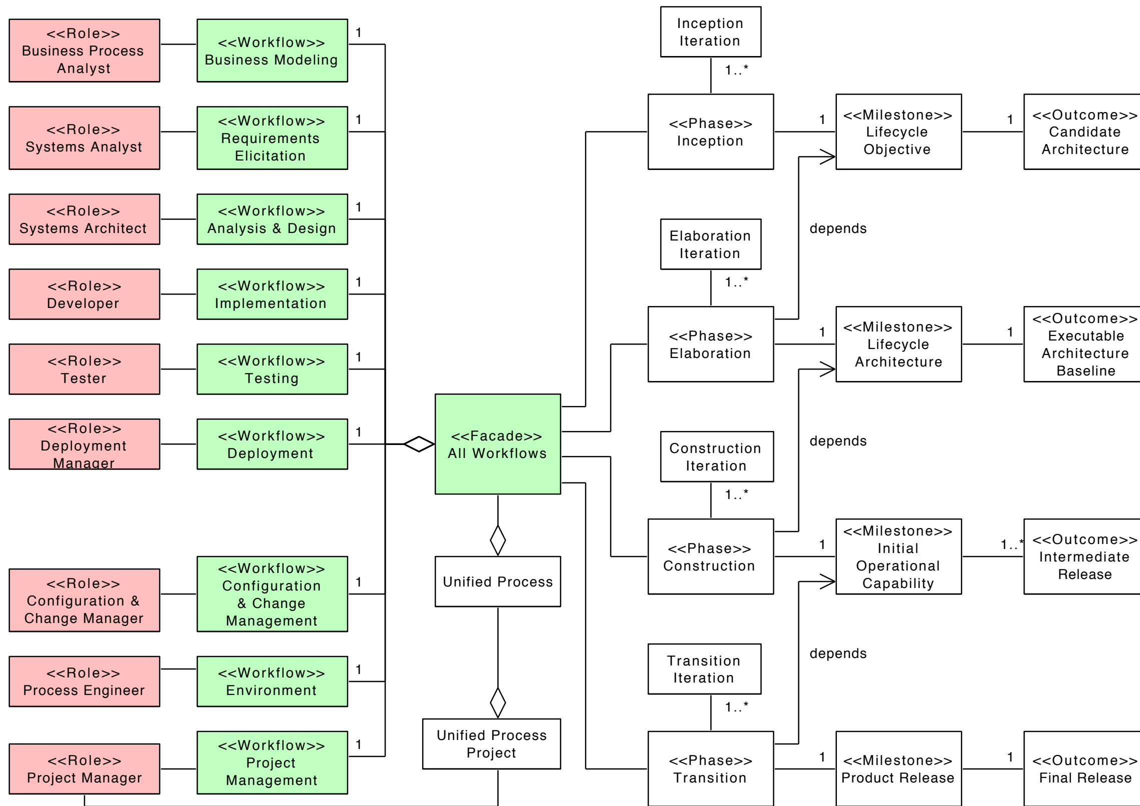


Figure 3.12: Static view of the Unified Process model with a UML class diagram as an example of an iterative instantiation of Rugby's process meta model (adapted from [JBR98])

Figure 3.11 shows the lifecycle of the dynamic model of the Waterfall process. Phases map to workflows. In each phase the full work load is solely on the corresponding workflow, all other workflows sleep.

3.5 Instantiation of Unified Process as Iterative Model

The second example of an instance of Rugby's process meta model is the iterative Unified Process that was first described by Jacobson et al. [JBR98]. The static view of the process model is shown in Figure 3.12. It includes the four phases inception, elaboration, construction and transition, six core workflows for business modeling, requirements elicitation, analysis, design, implementation, testing and deployment and three supporting workflows for configuration & change management, environment and project management. Each phase consists of multiple iterations and is connected to all workflows via the *All Workflows* facade. While the phases have a different emphasis on the workflows, all workflows run in parallel and perform work in each phase.

The *Inception* phase creates the *Candidate Architecture* as outcome and upon the *Lifecycle Objective* milestone, the *Elaboration* phase is activated that creates an *Executable Architecture Baseline* and lasts until the *Lifecycle Architecture* milestone. The *Construction* phase creates multiple *Intermediate Release*s, which are increments that include a specific functionality and build upon the last intermediate release. This is usually the longest phase in the Unified Process. It finishes when the *Initial Operation Capability* milestone is reached. Then the *Transition* starts with the goal to deploy the release in the target environment and lasts until the *Final Release* is produced.

The dynamic view of the process model is shown in Figure 3.13. Milestones are modeled as Rugby events, which transition between two phases. Phases can be split into multiple iterations. The inception, elaboration and transition phases run until their respective outcome is finalized. In the construction phase, each iteration creates an intermediate release as outcome.

At the beginning of the project the six core workflows and the three supporting workflows are activated. Each phase has a specific emphasis on certain workflows, e.g. the inception phase focuses on business modeling and requirements. Figure 3.14 shows the lifecycle of the dynamic model of the Unified Process.

The elaboration phase focuses on business modeling, requirements, analysis & design and implementation. In the construction phase, the workflows for business modeling, requirements and analysis & design reduce their work load, while the focus is on implementation and test. In later construction iterations, the workload of the deployment workflow becomes higher. In the transition phase, the focus is on deployment.

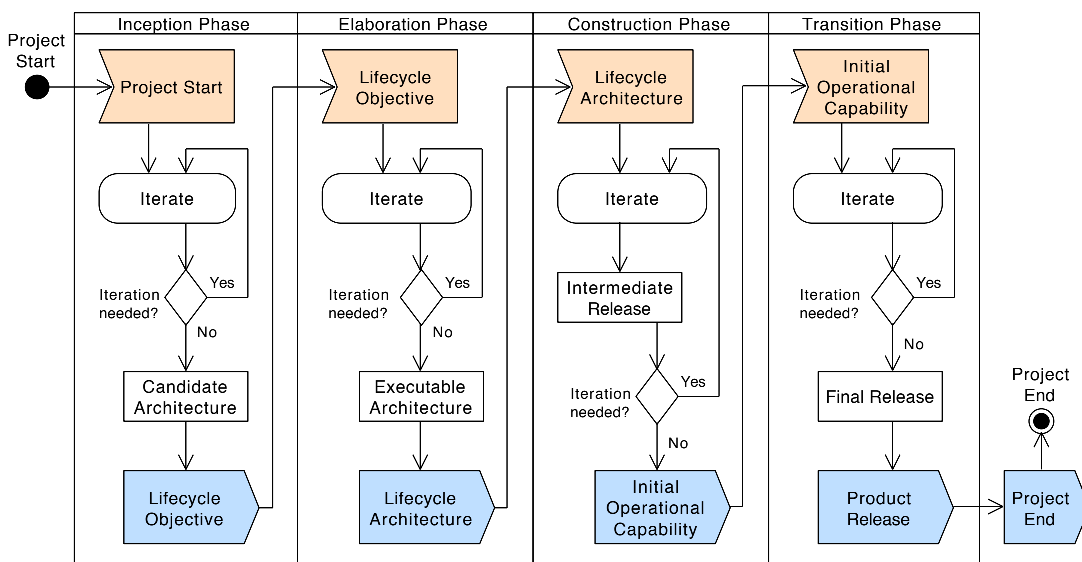


Figure 3.13: Dynamic view of the Unified Process model with a UML class diagram as an example of an iterative instantiation of Rugby's process meta model (adapted from [JBR98])

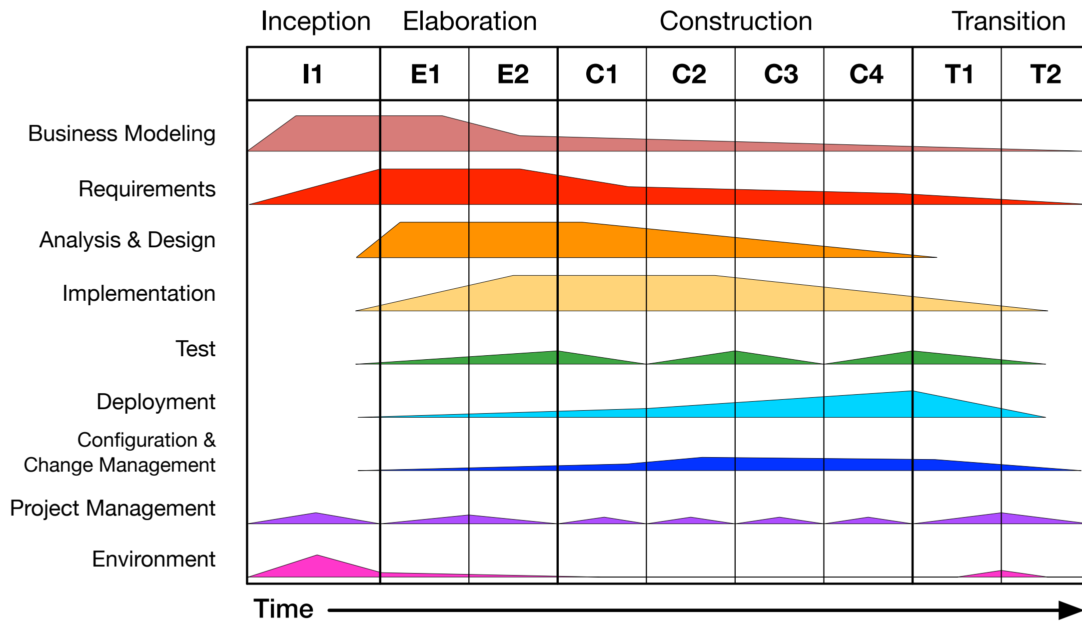


Figure 3.14: Lifecycle of the Unified Process as view of its dynamic model: Illustration of the relative emphasis of workflows in the Unified Process for different project phases (adapted from [Kru04])

The supporting workflows run with a small work load throughout the whole project. Only the environment workflow has a higher work load in the inception phase because tools need to be set up.

3.6 Instantiation of Scrum as Agile Model

The third example of an instance of Rugby's process meta model is the agile Scrum process that was described by Schwaber and Beedle [SB02]. The static view of the process model is shown in Figure 3.15. Its schedule includes sprints (modeled as instance of phase) that produce *Product Increment*s as outcomes, which are reviewed during the *Sprint Review Meeting*. Each product increment builds upon the previous one and incrementally adds new features. Scrum defines three roles: the *Scrum Master*, the *Product Owner* and the *Developer*. The Scrum Master is responsible that the development team follows the Scrum process. The Product Owner is responsible to define the product backlog by communicating with all stakeholders and creating, editing and prioritizing backlog items, so he is involved in the *Requirement Engineering* workflow. Developer build a self organizing and cross functional team that is responsible for the development workflows *Analysis*, *Design*, *Implementation* and *Testing*.

*Backlog Item*s are instances of *Work Item* and are managed in *Backlog*s which are instances of *Work Queue*s. Each sprint has a sprint backlog that includes all items that

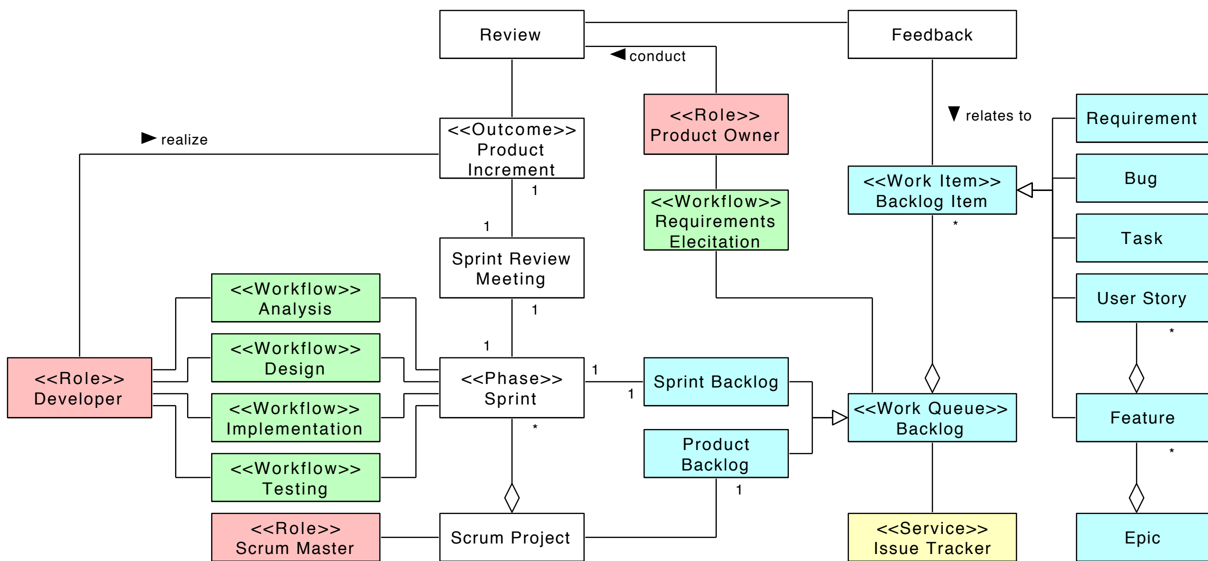


Figure 3.15: Static view of the Scrum process model with a UML class diagram as an example of an agile instantiation of Rugby's process meta model (adapted from [SB02])

the developers agree with the Product Owner to realize in the next product increment. The sprint backlog cannot change during a sprint. All other known backlog items are listed in the product backlog which can be changed by adding, removing, modifying or reprioritizing items.

The dynamic view of the process model is shown in Figure 3.16. After the project starts, the kickoff meeting is conducted, in which the Product Owner creates the product backlog as part of the requirements elicitation workflow. If the product backlog is finished, the event *Product Backlog created* leads to the start of the first sprint. Sprints are conducted until the project is finished.

A sprint starts with a *Sprint Planning Meeting*, in which Product Owner and development team negotiate the sprint backlog that is used as basis for the development. The team discusses status, impediments and promises in daily standup meetings while developing the backlog items. It creates the product increment, a time based release, before the sprint review meeting, which marks the sprint end. During the sprint review meeting, the Product Owner uses the product increment and gives feedback to the team in the usage workflow. A feedback report can lead to a revised product backlog, that is used for the next sprint planning meeting. Revising the product backlog is part of the requirements elicitation workflow.

Figure 3.17 shows a detailed dynamic view of the activity *Develop Backlog Item* and its surrounding elements. For each backlog item, the developers activate the four workflows analysis, design, implementation and test which are executed in parallel and influence each other. If a developer finishes a backlog item, the *Backlog Item finished*

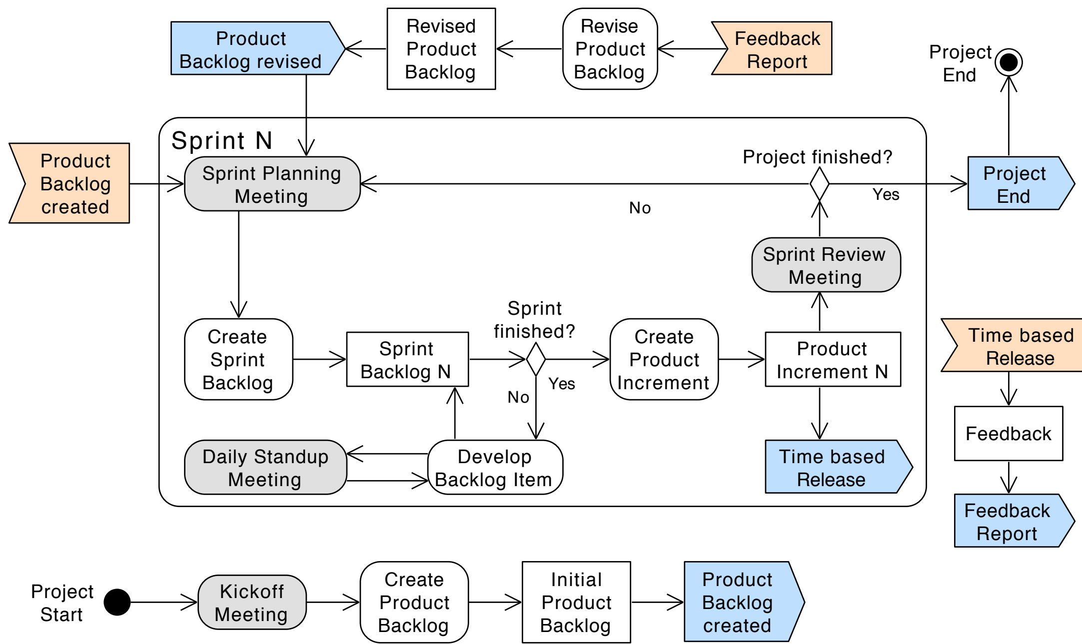


Figure 3.16: The dynamic view of the Scrum process model is an example of an agile instantiation of Rugby's process meta model (adapted from [SB02]). Grey activities are meetings (scheduled events).

event is created and the developer pulls the next backlog item from the sprint backlog until the sprint is finished. In between, developers meet regularly - usually on a daily basis - in the *Standup Meeting* and discuss status, impediments and promises.

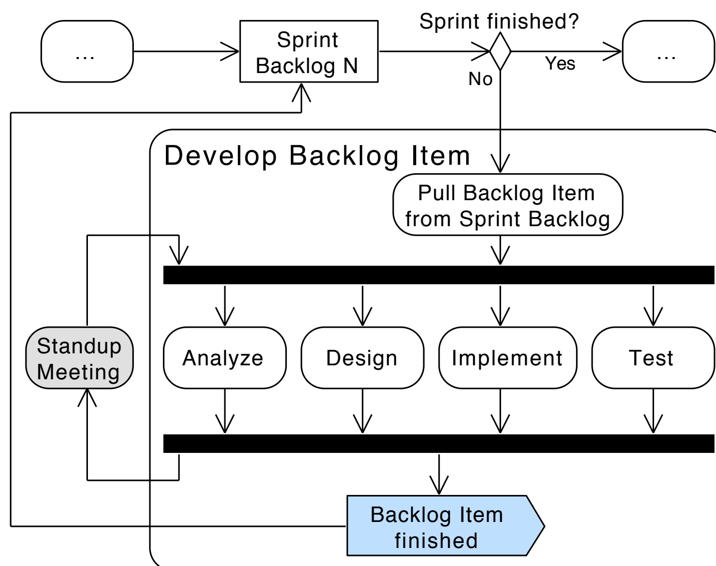


Figure 3.17: Details of the activity *Develop Backlog Item* in the dynamic view of the Scrum process model (adapted from [SB02]). Grey activities are meetings (scheduled events).

Figure 3.18 shows the lifecycle of the dynamic model of Scrum including six workflows. At the beginning of the project, there is a higher focus on requirements elicitation, analysis and design because the development team is not yet familiar with the application domain of the software. The average effort in these three workflows decreases over the time of the project when the developers get more familiar. Instead the work load in the implementation and testing workflows then increases. The project management workflow is most active in the beginning of the project when the most impediments occur and between two sprints, when the meetings for sprint review and sprint planning need to be organized and conducted.

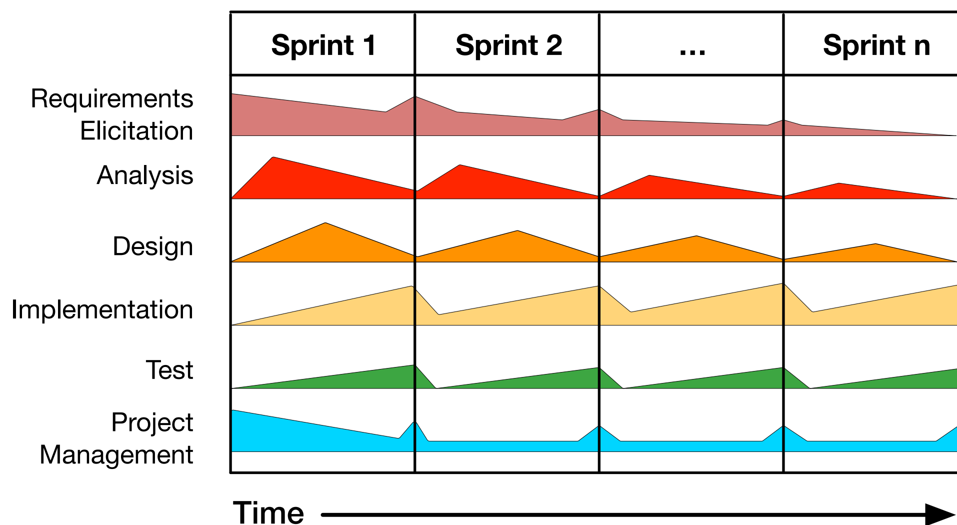


Figure 3.18: Lifecycle of the Scrum process as view of its dynamic model: Illustration of the relative emphasis of workflows in Scrum for different project phases (adapted from [SB02])

With these three examples, we have shown that different types of processes (linear, iterative, agile) can be modeled as instances of Rugby's process meta model.

3.7 Related Process Meta Models

In this section, we relate Rugby's process meta model to two other process meta models specified by the Object Management Group (OMG). The Software Process Engineering Meta model (SPEM) was released in version 1.0 by the OMG in 2002. The current version is 2.0 which was released in 2008 [Obj08b]. It is used to define and describe software development processes and their components.

As Rugby's process meta model, the SPEM includes a list of fundamental elements which are sufficient to describe different types of software process models. The scope of SPEM is limited to minimal elements necessary to define any development process.

In contrast to Rugby, it does not include specific features for particular domains or disciplines such as project management.

The goal of SPEM is to accommodate a large range of development methods and processes of different styles, cultural backgrounds, levels of formalism, lifecycle models, and communities. SPEM focuses on providing additional information structures for processes modeled with UML activities.

The SPEM 2.0 meta model is structured into seven main packages structuring the meta model into logical units: method plugin, process with methods, method content, process behavior, process structure, management content and core. While SPEM includes milestones as important events, it does not include a change model or an event taxonomy. Rugby's process meta model defines changes as events that can trigger the activation of workflows.

The Business Process Definition Meta Model (BPDM) has been specified in 2008 [Obj08a]. It is a standard definition of concepts used to express business process models. The meta model defines concepts, relationships, and semantics for the exchange of user models between different modeling tools.

BPDM extends business process modeling to include interactions between otherwise independent business processes executing in different business units or enterprises. These interactions are called choreographies and can be specified independently of its participants.

BPDM defines an event taxonomy and distinguishes between start and end events. It defines success events and failure events as normal end event types and abort events and error events as abnormal end event types in the standard model library. Similar to Rugby, BPDM allows modelers to extend the event taxonomy with newly defined events. BDPM events are similar to Rugby's change model where events trigger the activation of workflows. BDPM defines choreography as the synchronization between multiple organizations and orchestration as the synchronization between different departments of the same organization. However, BDPM does not include workflows related to software engineering.

Chapter 4

Rugby's Ecosystem

"Good process becomes invisible. It becomes culture."

— original author unknown

Based on Takeuchi and Nonaka [TN86], we use the term Rugby to describe a process that uses agile concepts of Scrum [SB02] and iterative workflows of the Unified Process [Kru04]. In this chapter, we describe Rugby's ecosystem[1] and the rationale behind it. We first show Rugby's top level design in Section 4.1, then define requirements in Section 4.2 and the use case model in Section 4.3. We describe static aspects of Rugby's process model in Section 4.4 and dynamic aspects of Rugby's process model in Section 4.5. We relate Rugby to the Agile Unified Process [CPP10] and the Disciplined Agile Delivery framework [AL12], both developed by Scott Ambler, in Section 4.6.

4.1 Top Level Design

Figure 4.1 shows releases and feedback bridging development and usage. Rugby's process model is based on the idea that frequency reduces complexity: the more often a developer performs an activity, the more common and the less complex it will become. Rugby follows Martin Fowler's recommendation "if it hurts, do it more often" [Fow11]. It uses this principle in three continuous workflows for review management, release management and feedback management.

Figure 4.2 shows the ecosystem of Rugby, divided into five environments. A developer interacts with the collaboration, development (the same as in Figure 4.1), integration and delivery environment, and a user interacts with the collaboration, delivery

[1] Highsmith defines an ecosystem as "a holistic environment" that includes several interwoven components: chaordic perspective, collaborative values and principles, and a barely sufficient methodology [Hig02]. We describe Rugby as an ecosystem of environments and workflows.

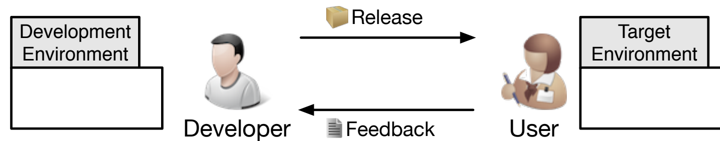


Figure 4.1: Shared understanding between developers and users through releasing executable prototypes and obtaining feedback

and target environment. Rugby focuses on the collaboration and delivery environments because they facilitate communication between developers and users.

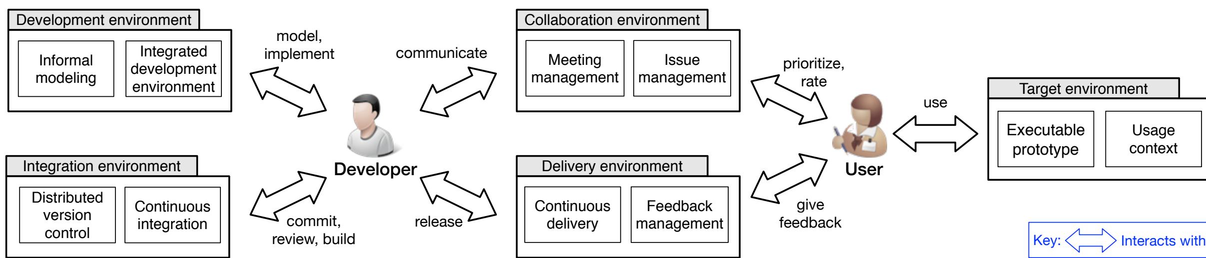


Figure 4.2: Top level design of Rugby's ecosystem: The developer interacts with four environments and the user interacts with three environments in Rugby (adapted from [BKA15])

A user is notified by the delivery environment if a new release is available and can then use the software in form of executable prototypes in the target environment. User feedback is uploaded to the delivery environment, connected with its release version and then forwarded into the collaboration environment. A user can prioritize and rate issues in the collaboration environment. Release management includes all activities concerning version control, continuous integration and continuous delivery. It focuses on the ability to release a change in the development environment as fast as possible into the target environment to the user. Feedback management covers the activities to motivate users to give feedback, to obtain it via structured channels and to include it back into the development environment.

4.2 Requirements

In this section, we describe functional and nonfunctional requirements for Rugby's process model respectively for tools that implement Rugby's workflows.

Functional Requirements

The following high level functional requirements (FR) describe Rugby's functionality, i.e. interactions between tools that implement Rugby and its actors or other exter-

nal systems [BD09]. They are independent of the actual tools implementing Rugby's workflows and are independent of the concrete instantiation of Rugby's process model. We group the functional requirements by the main workflows of Rugby, namely review management, release management and feedback management.

Review Management

We identified the following requirements for the review management workflow.

FR 1.1 Commit changes: The developer can commit changes (e.g. a bug fix or the realization of a backlog item) to the local repository in a specific branch. He can then synchronize these changes with the remote repository.

FR 1.2 Fix bug: The developer can fix a bug in the source code in the feature branch by committing changes to the feature branch (see FR 1.1).

FR 1.3 Create branch: The developer can create a feature branch in order to work separated on a specific backlog item.

FR 1.4 Update branch: The developer can update the feature branch with new changes from the development branch. He potentially has to fix merge conflicts (see 1.12).

FR 1.5 Delete branch: The developer can delete a feature branch on the local and remote repository, e.g. after the changes of the feature branch were merged to the development branch.

FR 1.6 Realize requirement: The developer can realize a requirement (more general a backlog item) in the feature branch by committing changes to the feature branch (see FR 1.1).

FR 1.7 Resolve review task: The developer can resolve review tasks with improvements that reviewers requested.

FR 1.8 Response to review comment: The developer can respond to review comments, e.g. by asking clarification questions.

FR 1.9 Improve source code: The developer can improve the source code according to the comments and suggestions of the reviewers by committing changes (see FR 1.1) to the repository. This automatically updates the merge request.

FR 1.10 Request merge into development branch: The developer can request a merge of the changes in a feature branch into the development branch, when he finished his work on the branch.

FR 1.11 Merge changes to development branch: The developer can merge the changes in the feature branch to the development branch after the corresponding merge request was approved by a defined number of reviewers and if no merge conflict occurs. If a merge conflict occurs, he needs to update the feature branch (see FR 1.4) and fix the merge conflicts first (see FR 1.12). Depending on the chosen solution for the merge conflict (see FR 1.12), it might be necessary for reviewers to review the updated merge request again (see FR 1.13).

FR 1.12 Fix merge conflicts: After updating the feature branch (see FR 1.4), the developer might need to fix merge conflicts by: (1) selecting his version; (2) selecting the other version; or (3) merging the changes of both version to a new version. After choosing one of these options, he needs to commit his changes (see FR 1.1) to the repository.

FR 1.13 Review quality: The reviewer can review the quality of source code changes in a feature branch (compared in a unified diff view to the version in the development branch) directly next to the source code.

FR 1.14 Request improvements: The reviewer can request improvements, if the source code, the architecture of the source code, or the design of the source code contains flaws, i.e. the source code does not adhere to patterns, best practices or coding guidelines defined within the development team.

FR 1.15 Write review comment: The reviewer can write comments next to the identified flaws in the source code to request improvements.

FR 1.16 Add review task: The reviewer can add review tasks for improvements to track the progress of the developer who implements the improvements.

FR 1.17 Approve merge request: The reviewer can approve the merge request, if the quality of the changes is sufficient and the realized backlog item in the feature branch should be integrated into the development branch.

FR 1.18 Decline merge request: The reviewer can decline the merge request, if the quality of the changes is not repairable or if the realized backlog item in the feature branch should not be integrated into the development branch.

Release Management

We identified the following requirements for the release management workflow.

FR 2.1 Configure build plan: The release manager can configure the build plan on the continuous integration server which includes setting up a connection to the source code repository and configuring build, test and delivery stages as well as execution build scripts for the used development environment.

FR 2.2 Integrate code: The continuous integration server can integrate the source code of the application, which is stored in the source code repository, by compiling, linking and packaging the source code.

FR 2.3 Execute test cases: The continuous integration server can execute test cases in the source code repository and create a test report which test cases pass and which fail.

FR 2.4 Notify about build status: The continuous integration server can notify the developer about the build status, e.g. via email.

FR 2.5 Build application: The continuous integration server can build the application which includes integrating the code (see FR 2.2), executing the test cases (see FR 2.3) and notifying the developer about the build status of the integration and test results (see FR 2.4).

FR 2.6 Detect changes: The continuous integration server can detect changes in the source code repository, in particular it can detect new commits (see FR 1.1) and new branches (see FR 1.3). Such changes automatically lead to a new build of the application (see FR 2.5).

FR 2.7 Upload build to delivery server: The continuous integration server can upload the build to the delivery server.

FR 2.8 Create release notes: The continuous integration server can create release notes for a specific build by accessing the issue tracker and the commit history of the source code repository.

FR 2.9 Edit release notes: The creator of the release (developer or release manager) can edit the automatically created release notes (see FR 2.8) and e.g. add a question about a specific aspect of the application that the user should answer by providing feedback (see FR 3.1).

FR 2.10 Select user groups: A release manager can select user groups on the continuous delivery server who will be notified about a release (see FR 2.16) and who can download and use the release (see FR 2.14 and 2.15).

FR 2.11 Create release: Developer and release manager can create releases. Rugby distinguishes between time based releases by the release manager (e.g. at the end of a sprint) and event based releases by the developer (e.g. when feedback is needed). The developer can create an event based release, even if the test cases fail (see FR 2.3). The release manager can only create a time based release if the test cases pass (see FR 2.4). The creation of the release includes in both cases the creation and editing of release notes (see FR 2.8 and FR 2.9), the upload of the builded application to the delivery server (see FR 2.7) and the selection of specific user groups (see FR 2.10).

FR 2.12 Demonstrate status: The developer can use an event based release to demonstrate the status of his work, e.g. in a meeting on a device or in a test environment.

FR 2.13 Promote release: The release manager can promote a release to a new user group. If a release was e.g. first delivered only to the developers group, the release manager can promote the same release to the customers group so that they can use it. In the same way, the release manager can promote releases from test environments to production environments. In a promotion, the same release is used and it is not necessary to create a new release.

FR 2.14 Download release: The user can download a release from the delivery server to his own device.

FR 2.15 Use application: The user can use the released application on his own device.

FR 2.16 Notify about new release: The continuous delivery server notifies the users in the selected user group about a new release, so that they can download and use the release (see FR 2.14 and 2.15).

FR 2.17 Configure user groups: The release manager can configure user groups on the continuous delivery server by adding and removing users and by editing existing users. These groups are the basis for the selection of user groups during a release (see FR 2.10).

Feedback Management

We identified the following requirements for the feedback management workflow.

FR 3.1 Provide feedback: The user can provide feedback directly in the application in form of a feedback report. The report is enriched with usage context (see FR 3.6) and uploaded to the continuous delivery server (see FR 3.4).

FR 3.2 Vote for feedback: The user can see existing feedback by other users and can vote for an existing feedback report to express his desire that the feedback is considered by the developers. This can help to reduce the number of duplicates and supports the developer to understand the priority of the feedback when he analyzes it (see FR. 3.13).

FR 3.3 Comment on feedback: The user can comment on existing feedback by other users to express his opinion on the existing feedback. This can help to reduce the number of duplicates and supports the developer to understand the feedback when he analyzes it (see FR. 3.13).

FR 3.4 Upload feedback: Feedback by the user is automatically uploaded to the continuous delivery server which stores the feedback in the issue tracker (see FR 3.5).

FR 3.5 Store feedback in issue tracker: The continuous delivery server automatically stores uploaded feedback in the issue tracker so that developers can be notified about it (see FR 3.16) and can analyze it directly in the issue tracker (see FR 3.13).

FR 3.6 Record usage context: Usage context is automatically recorded by the application and attached to the feedback report that the user creates (see FR 3.1) or to the automatically detected crash report (see FR 3.18). This information helps the developer to analyze the feedback (see FR 3.13) and to reproduce problems.

FR 3.7 Attach media: The user can attach images, voice recordings and videos to the feedback. These media attachments are stored as part of the usage context (see FR 3.6).

FR 3.8 Record environment: The application automatically tracks environment information such as the hardware configuration of the device, the operating system version, network conditions or the application version. This information is stored as usage context in the feedback report (see FR 3.6).

FR 3.9 Record screen: The application automatically tracks the screens the user has seen before he creates the feedback report or before crash reports are detected. This information is stored as usage context in the feedback report (see FR 3.6).

FR 3.10 Record stack trace: The application automatically tracks the stack trace in cases of crashes and exceptions. This information is stored as usage context in the feedback report (see FR 3.6).

FR 3.11 Record interaction steps: The application automatically tracks interaction steps such as button clicks or textfield inputs. This information is stored as usage context in the feedback report (see FR 3.6).

FR 3.12 Pull feedback: Developers can actively pull feedback by sending a new event based release (see FR 2.11) to a specific user group including a question in the release notes (see FR 2.9).

FR 3.13 Analyze feedback: Developers can analyze feedback in the issue tracker.

FR 3.14 Convert feedback into backlog item: Developers can convert feedback into backlog items in the issue tracker after they analyzed the feedback (see FR 3.13). They convert feature requests into requirements, bug reports into bugs and design requests into design improvements.

FR 3.15 Reply to user feedback: Developers can reply to user feedback and provide answers to user questions or comments to user problems. For instance, they can provide information about a functionality, if a user misses it although it is implemented. Additionally, they can ask clarification questions if they do not understand the user feedback.

FR 3.16 Reply to developer question: Users can reply to developers' clarifications questions, answer them and provide more information, e.g. in form of comments (see FR 3.3) and media attachments (see FR 3.7).

FR 3.16 Notify about feedback: The continuous delivery server notifies release manager and developers, if users report new feedback (see FR 3.1), if crash reports are detected (see Fr 3.16), and if the users comments (see FR 3.3) or votes (see FR 3.2) on existing feedback reports.

FR 3.18 Detect crash report: If the user uses the application (see FR 2.15) and a crash or an exception occur, the application automatically detects the crash respectively the exception. Usage context (see FR 3.6) is automatically attached to the crash report.

FR 3.19 Upload crash report: Detected crashes are automatically uploaded to the continuous delivery server including their usage context (see FR 3.6) if the user allows it.

Nonfunctional Requirements

Rugby is a process model that is adaptable for different organizations and different domains. Different organizations use different workflows and Rugby can be adapted to different project environments. We define adaptability as the capability of Rugby to adapt itself to changing circumstances. Rugby provides a common set of terms and concepts in the meta model which can be selected via tailoring before the project starts. So tailoring in Rugby is defined as the selection of workflows that are applied in a project.

Another adaption possibility is the customization of the chosen workflows within the projects, when activities of the workflow are changed. In addition, Rugby's change model with its event taxonomy is extensible to new domains. New events can be added to the project and workflows can be extended to subscribe to different change events and to generate different events. This allows the extension of Rugby to new domains such as education. Therefore, we state the following nonfunctional requirements (NFR) for Rugby.

NFR 1 Tailorability: The project manager can tailor Rugby's process to include new workflows or remove existing workflows from the process model before the project starts. New workflows can include new change events in the event hierarchy of the change model (see NFR 3).

NFR 2 Customizability: The workflow manager can customize a workflow during the project. He can adjust the workflow e.g. when developers find improvement possibilities in retrospective meetings or when other involved stakeholders request workflow changes. Customizing the workflow might also lead to new change events in the event hierarchy of the change model (see NFR 3).

NFR 3 Extensibility: Rugby can be extended to other domains, such as education, where different events occur, e.g. a student raising his hand and asking a question would be a new event. Therefore, Rugby allows to add new events to the event hierarchy of the change model, to change existing events or to remove events. Process tailoring (see NFR 1) or workflow customization (see NFR 2) might be necessary to react to new events.

4.3 Use Case Model

Figure 4.3 shows an overview of the roles inside the Rugby team that are used for the use case models in this section. As an additional role (not shown in Figure 4.3), we define the *User* (compare Figure 2.6) as a person who uses the delivered software application and who potentially provides feedback [Roe15]. Stakeholders in a Rugby project can have multiple roles. It is important to note, that a project manager or a Product Owner trying out the application to evaluate how a certain requirement was realized, also plays the role of a user. Therefore, it is possible that each team member plays the role of the user, in addition to his normally defined roles within the team.

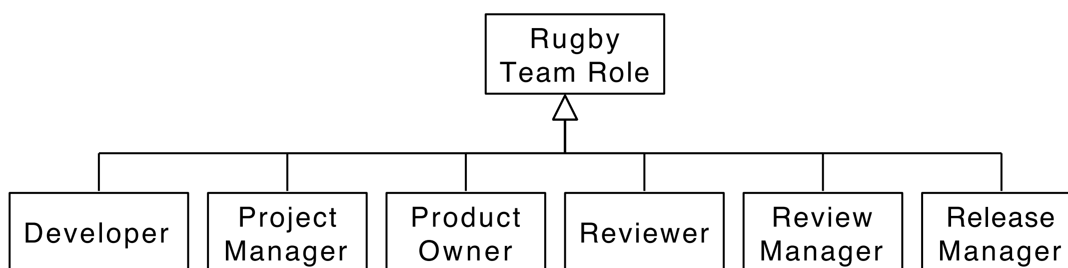


Figure 4.3: Team roles in Rugby (UML class diagram taxonomy)

Figure 4.4 shows the high level uses case model of Rugby. The project manager tailors the process and customizes workflows to the project and team environment. The review manager is responsible for the review workflow. The release manager is responsible for the release and the feedback workflow. The project manager conducts sprint planning and sprint review. The Product Owner and the project manager create the product backlog and the sprint backlog together.

Figure 4.5 shows the use case model of the review management workflow for the actors *Reviewer* and *Developer*. The use case model follows the requirements for review management and corresponds to the review model described in Figure 5.2 and to the review workflow described in Figure 5.4. The developer creates, updates and deletes feature branches to have separate places for the development of backlog items. All changes related to a backlog item are committed to the same feature branch.

When development is finished, the developer requests a merge to the development branch. The reviewer reviews the quality of the changes in the feature branch and requests improvements if necessary by adding review tasks and writing review comments. The developer addresses these comments by improving the source code and resolving review tasks. In addition, he can respond to review comments by asking clarification questions. As soon as the reviewer has approved the merge request, the developer merges it.

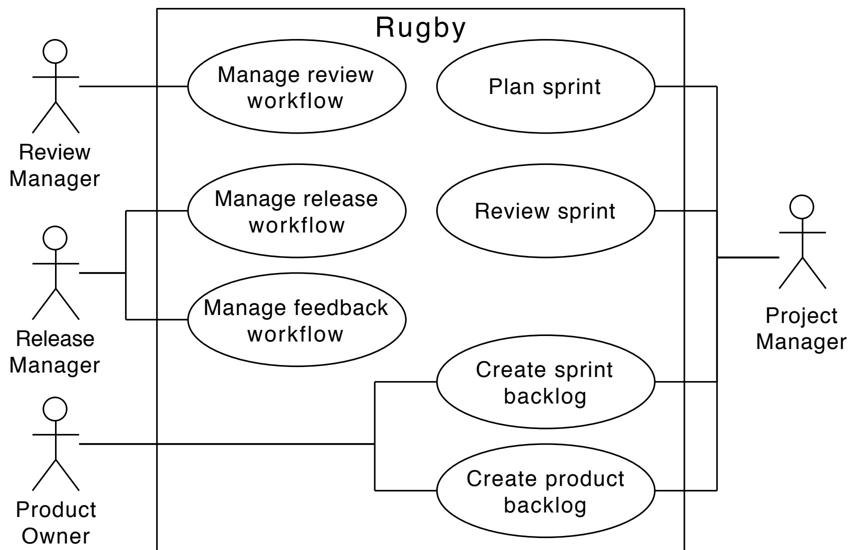


Figure 4.4: Rugby's high level use case model

Figure 4.6 shows the use case model of the release management workflow for the actors *Release Manager*, *User*[2] and *Developer*. The two external services *Continuous Integration Server* and *Continuous Delivery Server* are included as additional actors. The use case model follows the requirements for release management and corresponds to the release model described in Figure 5.9 and to the release workflow

[2]The user is defined as a role, that any stakeholder in the team can also have, compare Figure 2.6.

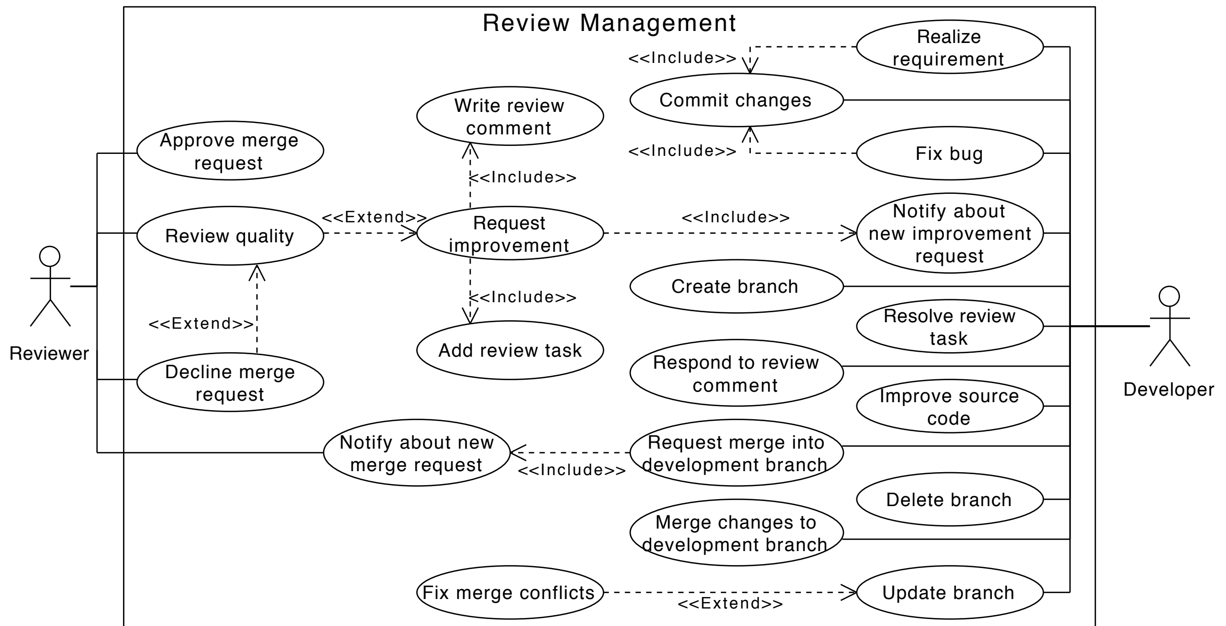


Figure 4.5: Rugby's review management use case model

described in Figure 5.10. Each commit of a developer is detected by the continuous integration server, which builds an application and executes test cases. The developer can use the created application to demonstrate his status in meetings or to obtain user feedback by creating an event based release directly from the feature branch. Then, the application is also created even if the source code was not reviewed yet and if test cases fail.

The creation of the release invokes the continuous integration server to upload the build to the continuous delivery server, to create release notes which can be edited and to notify the configured stakeholders about the new release. The release manager is responsible for configuring the build plan. This includes setting up continuous integration and continuous delivery server and customizing the integration, test and delivery stage of the build plan to the project environment. Before the sprint ends, the release manager is responsible to create the time based release from the development branch.

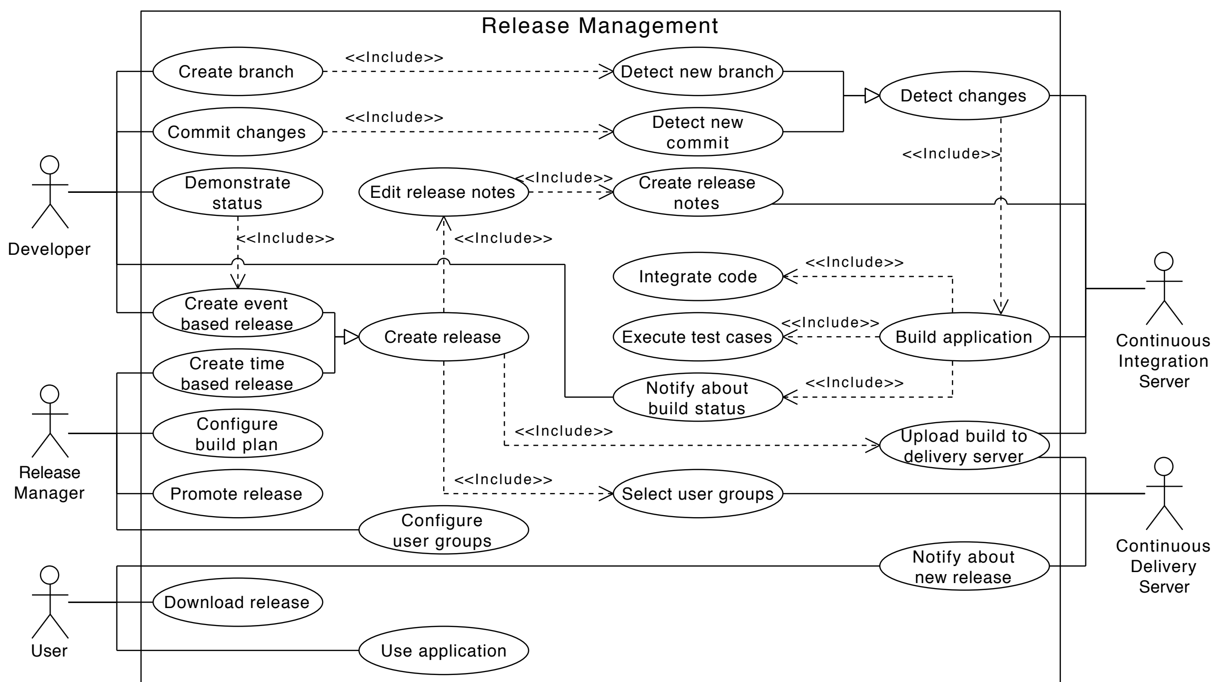


Figure 4.6: Rugby's release management use case model

Figure 4.7 shows the use case model of the feedback management workflow for the actors *User*[2] and *Developer*. The use case model follows the requirements for feedback management and corresponds to the feedback model described in Figure 5.12 and to the feedback workflow described in Figure 5.13. After an application was released, the user can give feedback by either creating a new feedback report or by commenting and voting on an existing report. The feedback system automatically records the usage context when feedback is provided including environment information such

as the operating system and network conditions. The current and potentially previous applications screens, the stack trace and the interaction steps are monitored and attached to the feedback report, so that the developer can understand the usage scenario. In addition the user can attach media such as screenshots, voice comments or videos.

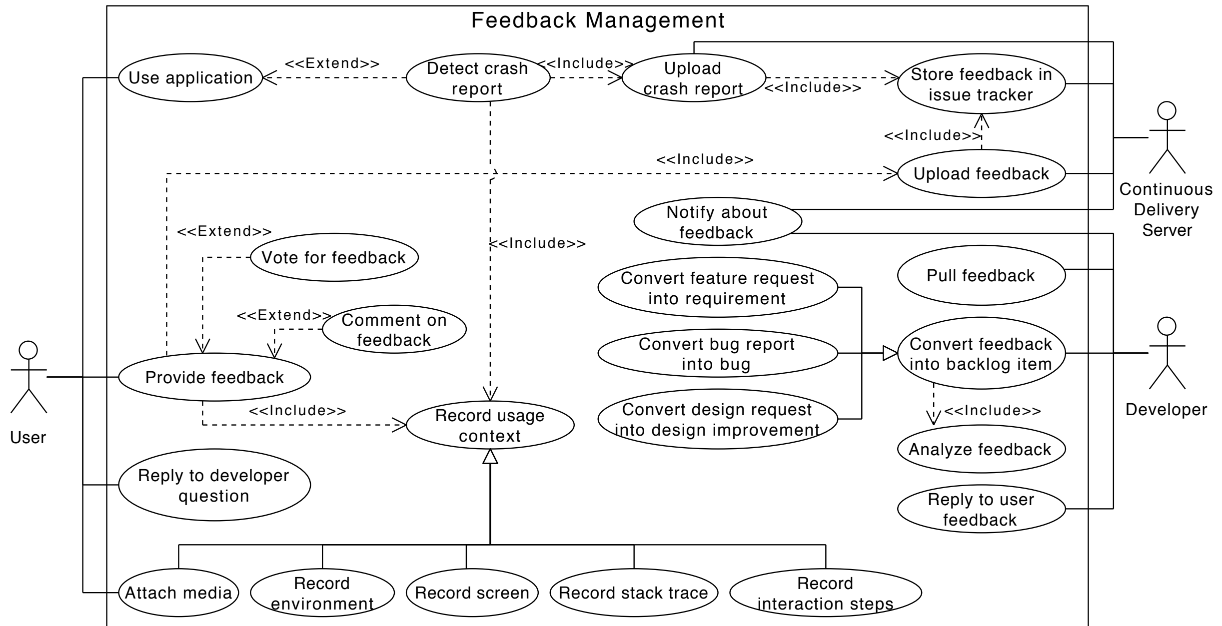


Figure 4.7: Rugby's feedback management use case model

The recorded usage context simplifies the analysis of feedback by the developer. If the report is still unclear or information is missing, the developer can reply to the user feedback and ask a clarification question. The user can then reply to the developer question and provide additional information. Depending on the feedback type, the developers convert it to backlog items. Feature requests are converted into requirements, bug reports and crashes are converted into bugs and design request are converted into design improvements. If a developer urgently needs feedback, he can pull feedback by sending an event based release including a particular question in the release notes.

4.4 Static View of Rugby's Process Model

Figure 4.8 shows the static view of Rugby's process model. It contains three packages for the review model, the release model and the feedback model. These packages contain the most important elements to show their relation to other elements of the process model. However, they do not include all details for readability reasons. We

show the corresponding full models in Figure 5.2 (review model), Figure 5.9 (release model) and Figure 5.12 (feedback model).

Rugby's process model includes a *Product Backlog* and a *Sprint Backlog* as defined in Scrum. In addition, there is *Demo Backlog*, which contains the backlog items needed to realize the demo of a release, e.g. at an intermediate presentation. The demo backlog is out of scope in this dissertation and is only shown for completeness reasons. It is part of the Tornado model and presented in [BKW12] and [XKB15].

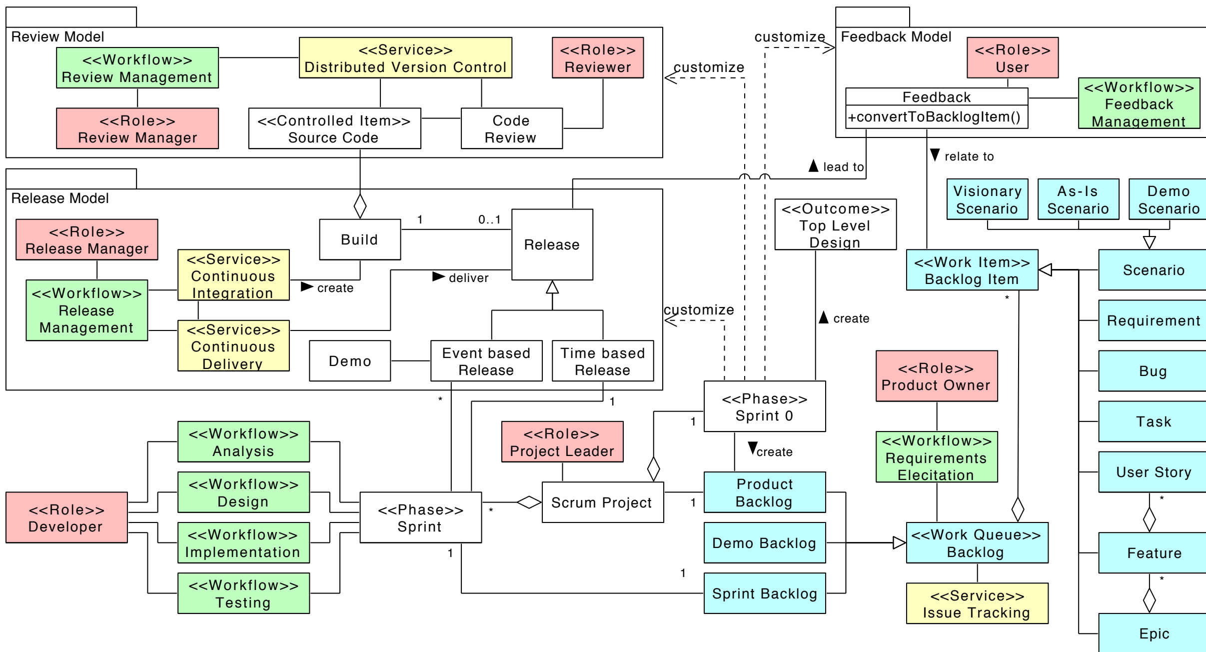


Figure 4.8: The static view of Rugby's process model shows project management concepts and their relation to the review, release and feedback model in a UML class diagram.

In Rugby, self organizing teams[3] develop software. Rugby focuses on innovation projects[4] where problem statements are formulated as visionary scenarios[5] and where requirements and technologies change during the project [BKW12]. In innovative projects, customers want to explore multiple ideas before they decide how vague requirements will be implemented concretely.

Rugby has two phases, an initial *Sprint 0* and development *Sprint*s as defined in Scrum. In Sprint 0, the team creates an initial version of the product backlog, deter-

[3]Self organizing teams are typical in agile projects. They manage their work on their own and are responsible for the outcome. While they still need mentoring and coaching, they do not require traditional managers to command and control.

[4]Innovation projects use new and immature technologies (hardware and/or software) and the project result might not be predictable. Therefore, empirical process control is needed.

[5]A visionary scenario describes a future system and is used in greenfield engineering and reengineering projects [BD09].

mines the top level design and releases it into an executable prototype called Release 0. In addition, the three supporting workflows for review management, release management and feedback management are customized to the project environment. Each development sprint leads to one time based release (comparable to a product increment in Scrum) and optionally to multiple event based releases in which the team can pull feedback from customers and users. Rugby's process model includes the following services which are provided by specific tools:

- *Distributed Version Control* allows to store source code and to review it before it is merged.
- *Continuous Integration* allows to regularly build and test source code.
- *Continuous Delivery* stores *Release*s[6] so that users can download them.
- *Issue Tracking* allows to manage backlog items, different backlogs and visualizes the status of items on a taskboard.

4.5 Dynamic View of Rugby's Process Model

A dynamic view of Rugby's process model is shown in Figure 4.9. Rugby defines the Sprint 0 (comparable to the elaboration phase of the Unified Process) as an upfront project phase which customizes the three workflows for review management, release management and feedback management. Another goal in Sprint 0 is creation of a shared understanding of the project requirements and the software architecture. Typical deliveries in this phase, which takes two to four weeks depending on the experience of the team, are an empty review, the Release 0 and an empty feedback to make sure the workflows are set up properly and everyone in the team is able to apply them. Additional goals are the creation of the initial product backlog, the top level design showing the high level software architecture and a first version of an analysis object model that shows the most important taxonomies and relationships in the application domain of the project. The top level design is manifested in the Release 0, which includes all major subsystems with one facade class that is invoked upon start and shows e.g. a welcome message.

After Sprint 0, the team works in development sprints as defined in Scrum and has to produce a product increment before the sprint review meeting. Development sprints in Scrum can be mapped to the construction phase in the Unified Process. The

[6] While a product increment is a time based release, that has to be produced at the end of each sprint, the development team also creates event based releases when they need feedback.

difference to Scrum is that in Rugby, the team can produce event based releases and obtain feedback during the development sprint.

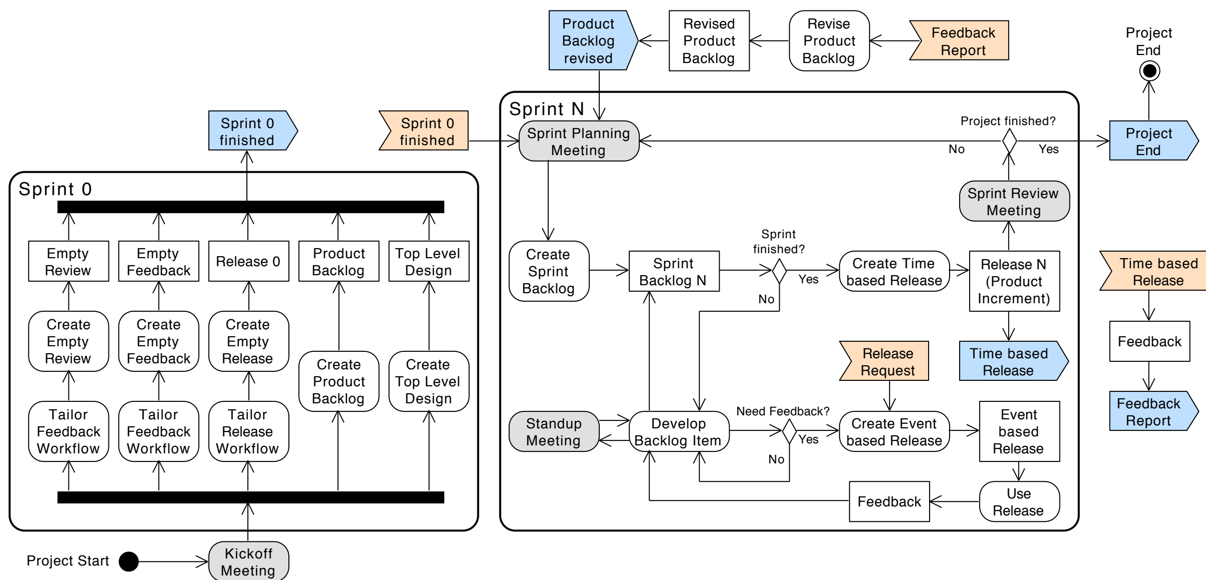


Figure 4.9: Dynamic view of Rugby's process model: UML activity diagram with the distinction between Sprint 0 and development sprints.

During the sprint planning meeting, the team baselines visionary scenarios to be realized in the upcoming sprint so that the sprint backlog includes a defined set of requirements. The customer specifies these requirements detailed enough so that the developers can start to work. However, he does not need to fully describe them, and he can still challenge developers to come up with own ideas how to realize a vague requirement and turn it into a product increment. Implementing a visionary scenario during the sprint might raise new questions for the team members, which they could not have thought of in the sprint planning meeting.

Presenting a first mockup for the visualization of a user interface could also lead to a requirement change because the customer might have different expectations that he was not able to express in words during the sprint planning meeting. Work that could be done during the same sprint would shift to the next sprint if customer collaboration and changing requirements during a sprint would be disallowed. In difference to Scrum, Rugby allows that requirements are further discussed and negotiated within the sprint.

Developers can create event based releases with partially developed functionality, when they want to obtain feedback from users. We call such releases event based to highlight the difference to the usual time based product increment at the end of the sprint. Event based releases help to illustrate the current realization of a requirement and to obtain feedback whether the developer is on the right track. The team does

not have to wait until the end of a sprint to deliver software to the customer and can save time to increase the quality of the product increment. If additional feedback is implemented in a sprint, the team can decide to move other backlog items to the product backlog to be realized in the next sprint. Candidates are the items with the lowest priority which the team did not yet start to realize. When negotiating about the inclusion of feedback and moving backlog items to next sprint, estimates can be used as a currency so that the effort within one sprint stays roughly the same.

Figure 4.10 shows the details of the activity *Develop Backlog Item* and surrounding actions in Rugby. Before the changes to the source code of one backlog item are merged, a code review takes place, where reviewer request improvements, if the quality is not sufficient. Feedback that was obtained by event based releases influences the development of a backlog item.

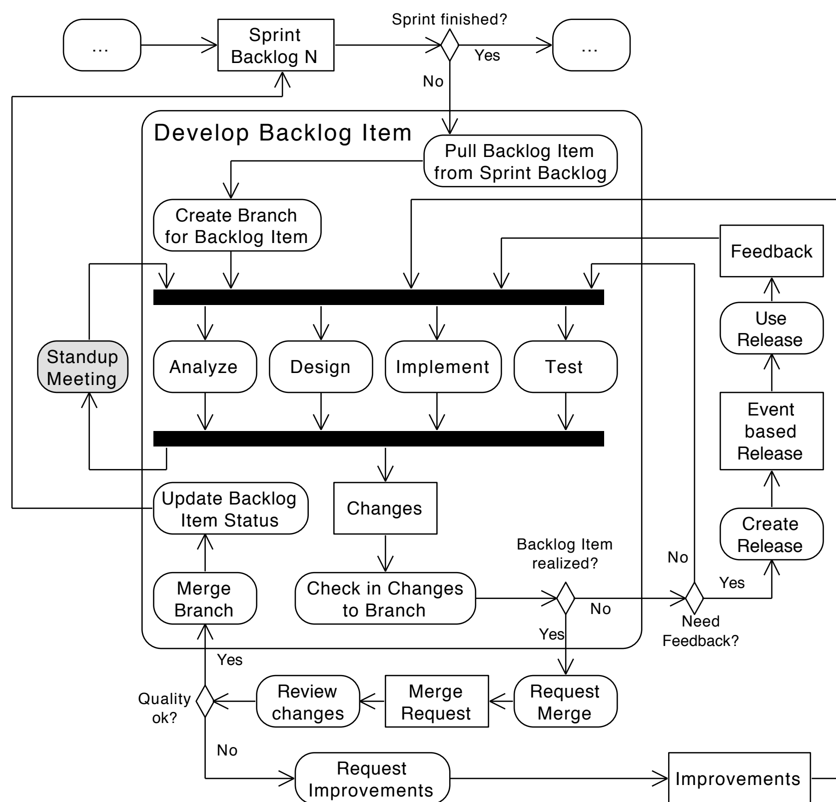


Figure 4.10: Dynamic view of Rugby's process model: Details of the activity Develop Backlog Item

Figure 4.11 shows the synchronization of parallel workflows in Rugby through Rugby change events. The Product Owner prioritizes *New Backlog Item*s that are identified through change requests or feedback reports and specifies the acceptance criteria so that these items are ready for development. This might invoke the development workflow, where developers analyze, design, implement, and test in parallel to produce

source code changes. If the backlog item is realized, i.e. it fulfills all acceptance criteria, the developer requests a merge. This activates the review management workflow where the reviewer either accepts the changes so that they can be merged, or where the reviewer requests improvements, so that development is activated again.

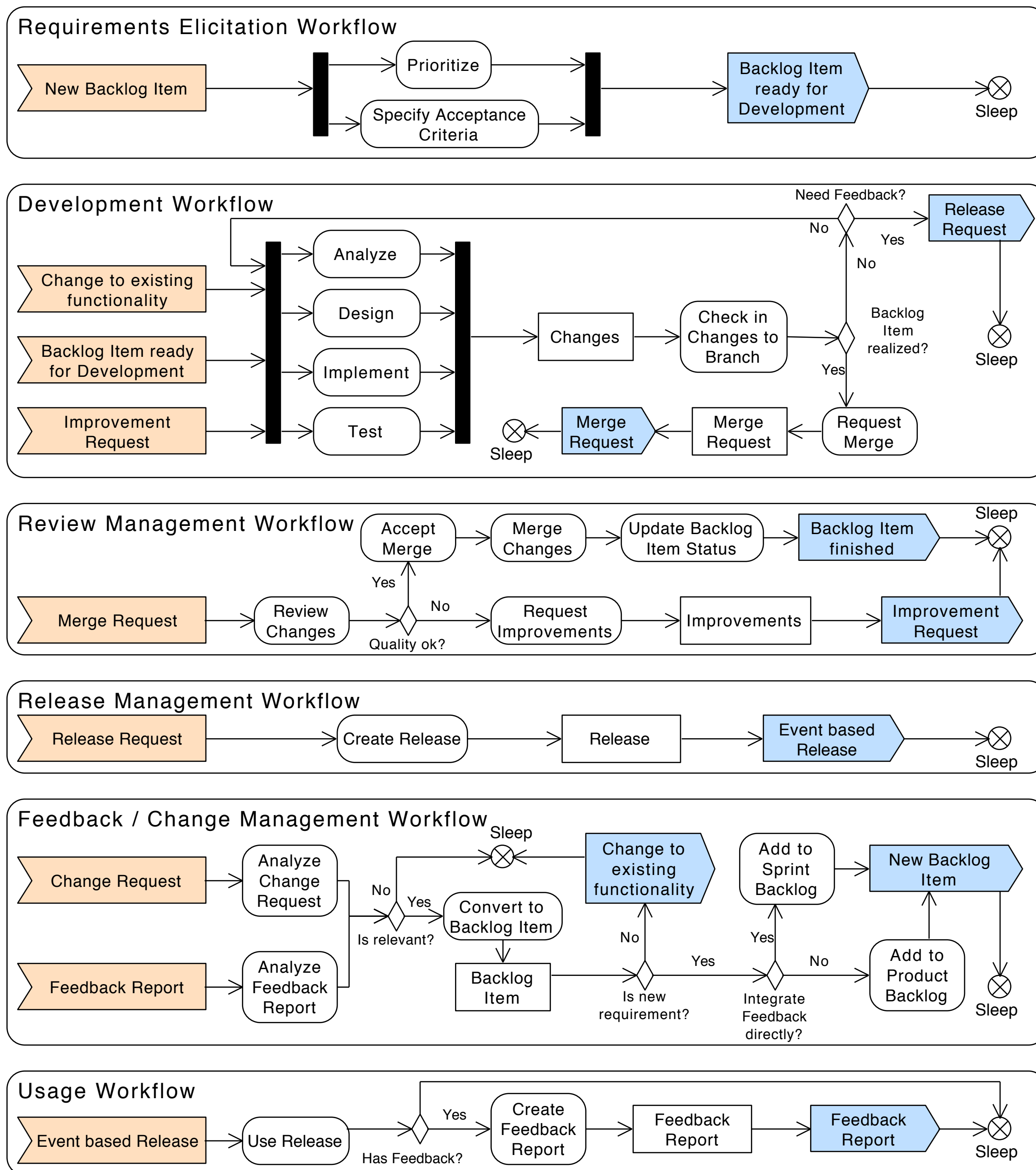


Figure 4.11: Dynamic view of the synchronization of parallel Rugby's workflows through change events

Another possibility is that the developer requests a release of the non finished backlog item in case he needs feedback or has a question. Then the release management workflow is activated and an event based release is created which is used by a user in

the usage workflow. If the user has feedback, he creates a feedback report, which is analyzed in the feedback management workflow and which either leads to a change to an existing functionality or to a new backlog item.

4.6 Related Process Models

Scott Ambler described the Agile Unified Process (AUP) [CPP10] as a hybrid model combining the strengths of the Unified Process and agile methods. The AUP applies techniques such as test driven development, agile modeling, change management, and database refactoring. It includes seven disciplines, in particular configuration management and deployment, which are similar to Rugby's release management workflow.

As shown in Figure 4.12, it distinguishes between development releases that are deployed to the quality assurance stage and production releases that are deployed to the production stage. Rugby also distinguishes between internal and external releases. Similar to Rugby, the AUP reduces the difficulty of release management through an increased frequency of releases with smaller changes. However, it does not describe concepts related to review management such as branch and merge management which are important elements of Rugby. It also lacks a similar approach for event based releases and does not include feedback management.

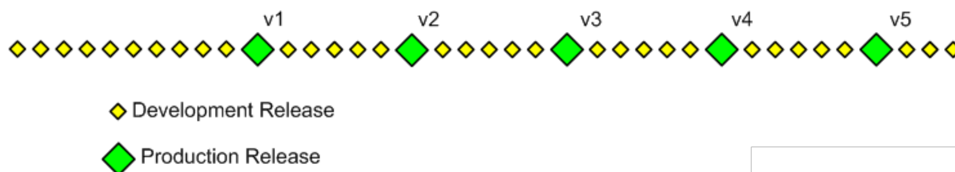


Figure 4.12: Agile Unified Process Timeline with the distinction between development releases and production releases (adapted from [CPP10])

The AUP was superseded by its successor Disciplined Agile Delivery (DAD) which was also developed by Scott Ambler and is currently in version 2.0 [AL12]. DAD is a process decision framework for enterprise IT around incremental and iterative solution delivery and is a means of moving beyond Scrum. It includes an agile delivery lifecycle as shown in Figure 4.13 and claims to include elements of Scrum, Extreme Programming and agile modeling.

It is similar to Rugby in having an upfront inception phase where initial requirements and the release plan are developed as a list of work items. These work items are described in more detail later in the project. It also describes enhancements requests and defect reports as feedback which influences work items after the software was release

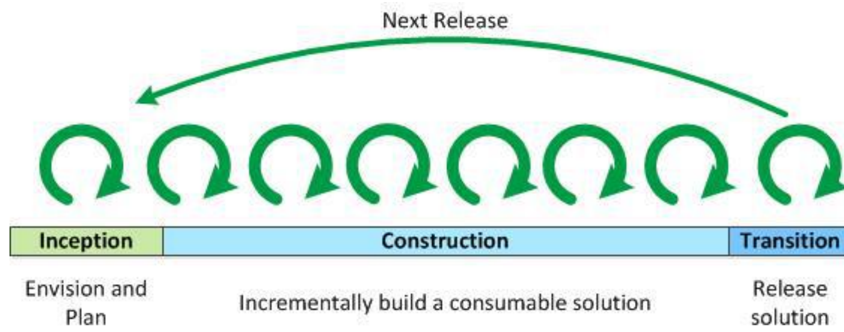


Figure 4.13: Disciplined Agile Delivery Lifecycle (adapted from [AL12])

into production. Similar to the Unified Process and Rugby, it describes workflows that span over the whole project, however it does not include a change model that triggers the activation of workflows.

Chapter 5

Rugby's Workflows

"The single biggest problem in communication is the illusion that it has taken place."

— George Bernard Shaw

In this chapter, we describe Rugby's workflows, show static and dynamic views of their process models and describe related work.

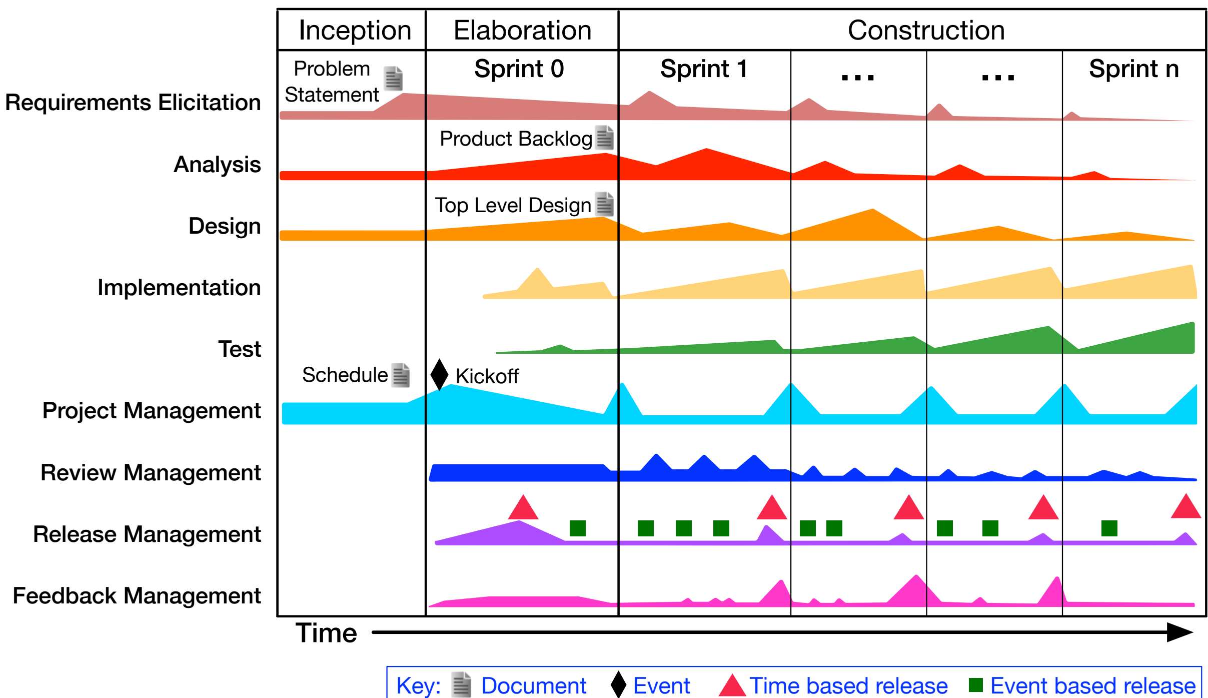


Figure 5.1: Rugby's lifecycle as view of its dynamic model: It shows the amount of work as colored areas for each workflow in different phases (adapted from [Kru04, BRS09, BKW12]).

The lifecycle in Figure 5.1 is a view of Rugby's dynamic model. It visualizes the parallel running workflows[1] of Rugby's meta model and is adapted from [BRS09] and [BKW12]. The average effort of a workflow during a particular phase is visualized as the colored area in its horizontal bar.

Sprint 0 is comparable to the elaboration phase in the Unified Process: it focuses on the instantiation of Rugby's tailored process and its customized workflows, e.g. by creating an initial empty release R0, which is comparable to the executable architecture baseline in the Unified Process. It also requires the creation of the product backlog to collect all functional requirements for the construction phase (development sprints).

Rugby's lifecycle includes three supporting workflows: Review management enables continuous reviews including a branching model and merge requests which prevent poor code with design flaws from entering the main codebase. Section 5.1 describes the review management workflow and Section 5.2 relates it to other work in the area of code reviews.

Release management enables continuous delivery including event based releases which enable developers to obtain user feedback within the Sprint. Section 5.3 describes the release management workflow and Section 5.4 describes related work in the area of release management.

Feedback management enables continuous feedback including a semi automatic mechanism to handle feedback and crash reports to decrease developer effort in analyzing feedback and increasing user motivation to provide feedback. Section 5.5 describes the feedback management workflow and Section 5.6 describes related work in the area of user feedback.

[1]We adopted this idea from the Unified Process [Kru04].

5.1 Review Management Workflow

We defined an informal review workflow for source code collaboration and code reviews with the following goals:

1. **Early stage reviews**: The code is reviewed from the beginning of the project to adapt the fail early principle and to learn from mistakes as soon as possible.
2. **Continuous reviews**: Reviews are conducted regularly to guarantee high quality in the main codebase.
3. **Review responsibility**: Students conduct the review themselves to improve their learning experience and to reduce the effort for the instructors.
4. **High quality releases**: Only reviewed code is integrated to the main codebase and is present in product increments.
5. **Efficient reviews**: Each change is only reviewed once before it is integrated.
6. **Fast development process**: Reviews do not slow down the development process and the ability to release new features quickly.
7. **Customizability**: The workflow should be customizable to different project environments.

Figure 5.2 shows Rugby's review model which is based on distributed version control using the possibilities of branch and merge management. A *Merge Request* is initiated by a *Developer*, includes changes of one source *Branch* that should be merged into one destination *Branch* and consists of at least one informal *Review* by a *Reviewer*. The source branch is typically a *Feature Branch* in which a *Requirement* was realized in multiple *Commit*s by *Developer*s who want to integrate the *Modified File*(s) into the *Development Branch*.

Rugby uses a simplified version of git flow [Dri10] including merge requests as shown in Figure 5.3. A quality gate is Developers realize backlog items such as requirements on feature branches, use a development branch for the integration of realized requirements and a master branch to store released versions. When implementing a new feature, developers create a new feature branch and commit all related changes into this branch. Meanwhile other developers may work on other feature branches and may have already integrated their changes back into the development branch. The developers then need to pull these changes and merge them into their feature branch. They should regularly check if there are new changes on the development branch that they did not yet integrate into their feature branch.

The parallel development of features in separate branches might lead to merge conflicts. Therefore it is necessary to keep the backlog items, which are realized in

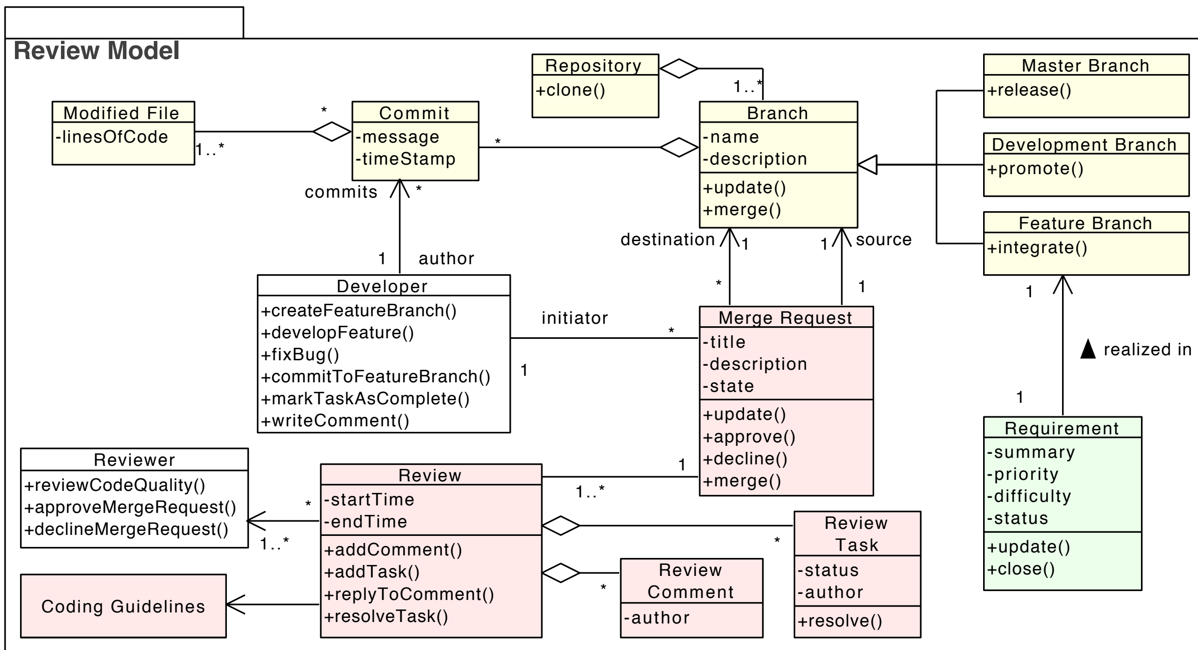


Figure 5.2: Rugby's review model: concepts of the review workflow and their relations in a UML class diagram - colors highlight related concepts

features branches, small and to limit the lifetime of features branches. A feature branch should not exist longer than some working days, ideally only one or two working days. In addition, the development team should not work on too many feature branches at the same time and limit the work in progress to a few backlog items as defined in Kanban [And10]. One approach could be that between two and three developers work on the same feature and at most half as much feature branches exists as developers.

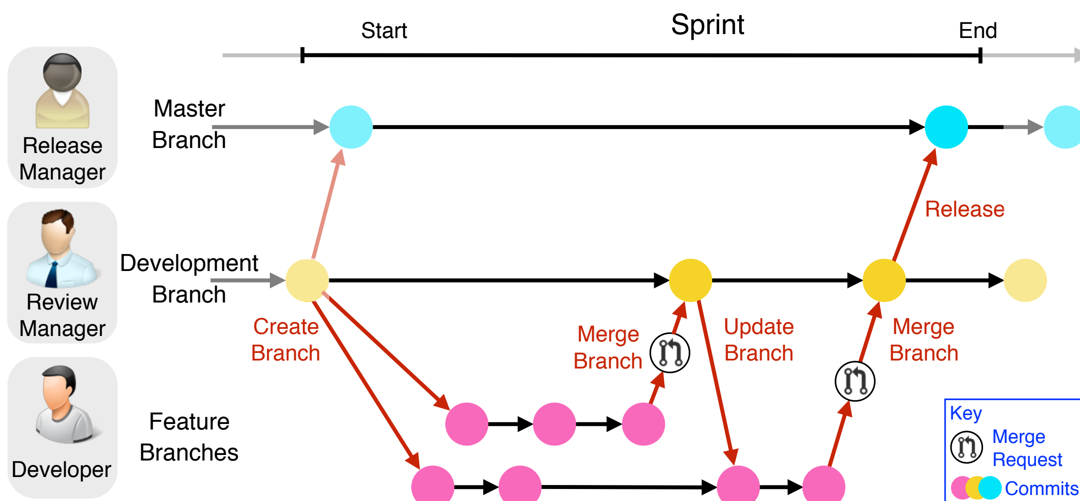


Figure 5.3: Rugby's branching model: usage of feature branches by developers and usage of merge requests (adapted from [Dri10])

When developers have realized a feature, they run unit tests locally and on the continuous integration server (not shown in Figure 5.3). If all tests pass, they request a merge into the development branch to integrate their changes. Merge requests are also called *pull requests* and are supported by several platforms such as GitHub, BitBucket and Stash. They act as a quality gate that prevents code with bad quality from being integrated into the main codebase in the development branch. When a developer files a merge request, he is requesting that the review manager accepts and then integrates the changes from the feature into the development branch. This review workflow is shown as UML activity diagram in Figure 5.4.

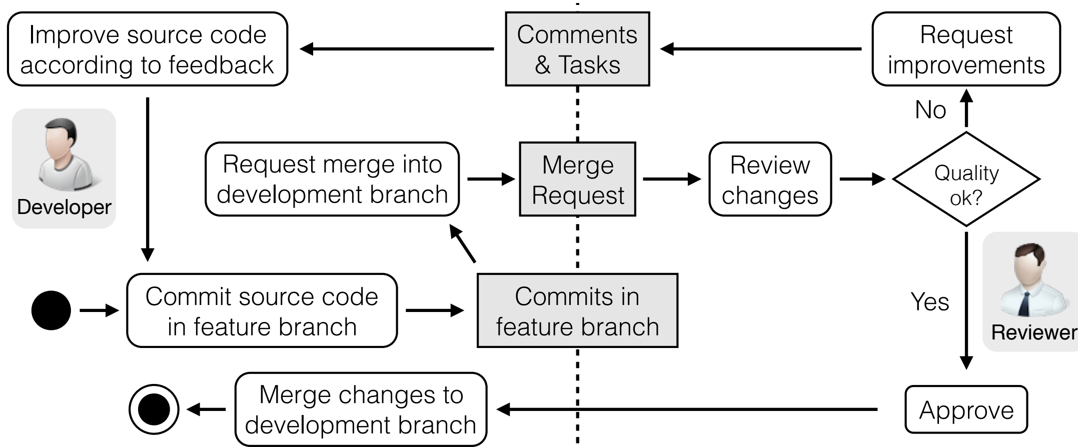


Figure 5.4: Overview of the review workflow with developer and reviewer as roles and their activities in a UML activity diagram (adapted from [KBB16])

During the quality gate, reviewers address the following questions:

- **Design Traceability:** Is the code traceable to the specified system and object design? Does it fulfill design principles such as low coupling and high cohesion?
- **Use of Patterns:** Does the code contain design or architectural patterns? Does it avoid software development antipatterns [BMMM98]? Does it avoid code smells?
- **Maintainability:** Does the code adhere to coding guidelines? Is it easy to read and understand?
- **Review History:** Does the code address feedback of previous reviews?

Merge requests show the accumulated changes of the feature branch. Only if the code meets defined criteria, reviewers approve the request. Thus, the workflow prevents feature branches with poor code quality or bad architectural decisions from being merged with the development branch. Problems or misunderstandings found by reviewers cause a comment added directly to the changes. The developer, who had

requested the merge, reads these comments and improves the code in response commits. The merge request is then updated automatically.

When all comments are addressed, reviewers approve the request. Compared to other collaboration models, this solution for sharing and reviewing commits creates a streamlined workflow. While git could send notification emails with a simple script, it becomes haphazard when developers discuss changes and have to rely on email threads, in particular when response commits are involved. Merge requests put a discussion platform on top of commits and branches into a web interface next to the repository. The branch based code review workflow has several advantages:

1) Only changes in the feature branch must be reviewed. If the change set of a feature branch is small, the workload for reviewing is small.
2) If an experienced programmer reviews the code for errors, there will be less defects in the code [CW00].
3) Developers prevent "broken windows" in the development branch, if they use this workflow from the beginning: "Don't leave broken windows (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered" [HT00]. Conducting code reviews avoids that bad design and poor code are distributed to the whole development team, and are potentially being reused in other places in the system. This alleviates the risks of the broken window theory in programming.
4) The workflow increases collaboration and knowledge transfer between developers, because merge requests facilitate conversations about actual source code. This improves peer learning [BCS14]. Inexperienced developers can learn best practices and coding guidelines while doing asynchronous pair programming [WK02]. This is especially helpful for balanced teams with beginners and advanced programmers. While merge requests allow for asynchronous pair programming, developers should also build pairs for synchronous pair programming [BA04].

However the workflow also poses challenges. If there are too few experienced developers who have to review many merge requests, they shortly become a bottleneck for the development progress. Another problem occurs, when features are too large and need several days or weeks to be finished. Then, the change set is large, so that reviewing and improving the code need a lot of time. In such situations the likelihood is high, that merge conflicts occur, in particular when parallel feature branches have overlapping changes. To alleviate such problems, features should be small. Additionally, merge requests can be integrated into a continuous integration system. When a

merge request is created, the integration server detects it, checks if a merge is possible without conflicts and if the merged code builds and all tests pass.

Detailed Review Workflow

We show workflow diagrams for each of the four activities of informal code reviews shown in Figure 2.4 and describe the details of each activity [KBB16]. While we show a standard workflow for each activity, the activity itself can be customized to the project environment. One example of customization would be a development team that does not use a branching model. Then it can e.g. switch to commit based code review, adapt some activities and still benefit the results of having a code review before source code changes are merged to the main code base.

The review workflow starts with the **preparation** activity, when the developer creates the source branch, as shown in Figure 5.5. Next comes the development work for the feature. Both during the implementation and throughout the rest of the workflow, the developer should be aware of new, relevant commits in the development branch and pull them to update his branch, as shown in Figure 5.3. The merge may result in conflicts, which are resolved locally before the workflow is continued. Once the developer finished the work, he commits to the branch, resulting in one or more initial commits, depending on the feature size. Typically, the commits are immediately pushed to the remote repository for storage, however, this step can be delayed until the merge request.

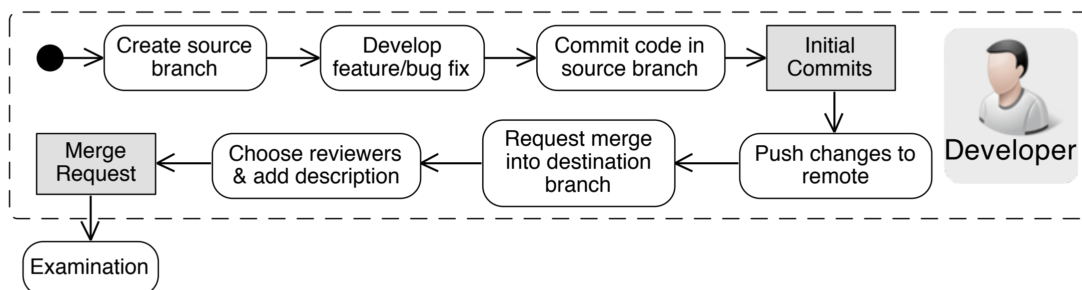


Figure 5.5: Workflow for the preparation activity in branch based code reviews (adapted from [KBB16])

When the developer implemented the feature, he requests a merge into the development branch. For the merge request to be created, the developer needs to select reviewers and add a brief description of the changes and their purpose. At least one experienced reviewer that is familiar with the purpose of the changes should be included among the participants. Nevertheless, the merge request is open to the rest

of the team, and any other member that would like to review the code, learn how to conduct reviews or simply is curious can join and contribute to the review as well.

Once the developer has completed the above steps, the **examination** activity begins that is shown in Figure 5.6 following either preparation or rework. The first point of interest is ensuring that the code was able to build successfully and passed the test cases. If either the built or a test case failed, the developer is notified and is expected to fix the code and update the merge request before the reviewers can examine it. Usually, this step is automated by using a continuous integration server and an integrated notification mechanism such as email.

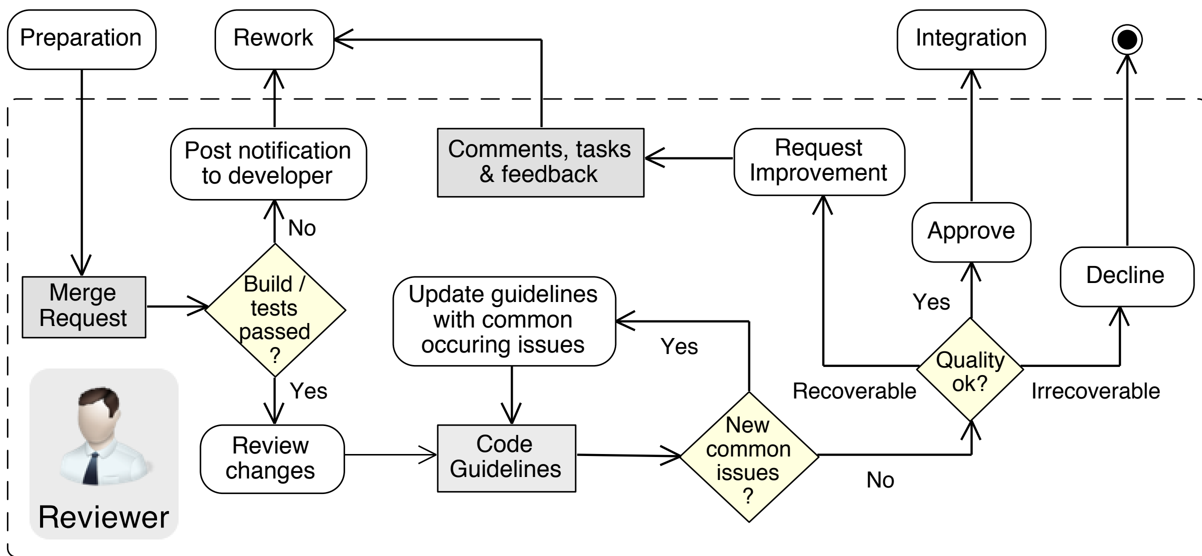


Figure 5.6: Workflow for the examination activity in branch based code reviews (adapted from [KBB16])

If the build is successful and all test cases pass, the actual review takes place. The assigned reviewers are automatically notified about the pending request and can follow a link to begin reviewing. Their goal is to find anomalies by answering the questions mentioned above. Since branch based reviews are asynchronous over the Internet, the reviewers conduct the examination independently of each other. When an anomaly is identified, it is documented using comments or tasks, which directly reference the code. Comments support discussion threads, whereas tasks are used to track anomalies that must be refactored.

If the examination follows the preparation activity, the reviewers have to examine all changes, however, if it comes after rework, they focus on the detected anomalies. The reviewers check if their prior feedback was implemented correctly and ensure that no other problems were introduced. The code guidelines are an essential source for the reviewers. A good opportunity to continuously improve and keep them relevant, is to update the document with common anomalies found after each review.

Once a reviewer has examined the changes, he decides whether the source code fulfills the team's quality standards. In the best case, the merge request is directly approved and the workflow proceeds to integration. However, if the reviewer believes that code quality should be further improved through rework, he requests improvements by noting the anomalies in comments and tasks. The third possibility is to outright decline the merge request, either because the developer is unwilling or incapable of implementing the feedback, or because the code quality is so poor that the improvement cost outweighs the benefit of having the work.

Declined merge requests, though, are very rare and should be treated as an indicator for a bigger problem concerning work allocation or lack of motivation. The more common case are recoverable merge requests, where with one or more improvement cycles of examination and follow up, the code reaches a state where it satisfies the quality requirements for approval. The developer is immediately notified about incoming feedback. Ideally, he first responds to comments that require clarification or discussion threads about possible or alternative solutions.

Once these are resolved, he starts the **rework** activity shown in Figure 5.7. He then improves his source code according to the feedback documented in tasks and comments. This includes fixing bugs, adjusting coding style to guidelines and restructuring the design to fit to the system architecture. If antipatterns or code smells were detected, the developer must implement refactored solutions or refactor the code smells. The second case for entering rework is when the merge request has a build or test case failure. In this context, the developer needs to find and fix the problems before the reviewers have a chance to examine his changes.

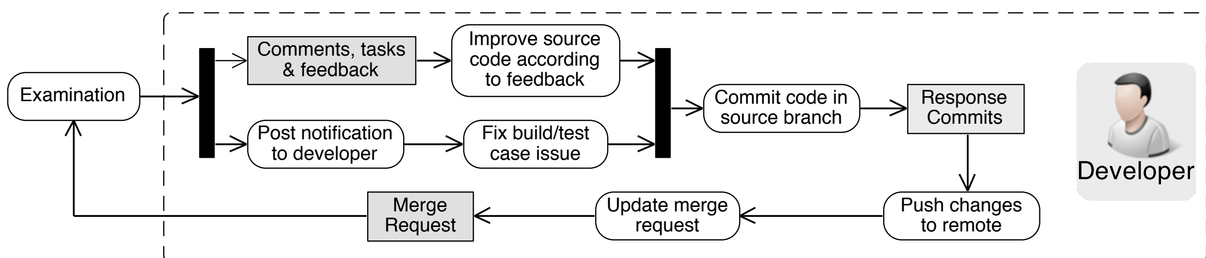


Figure 5.7: Workflow for the rework activity in branch based code reviews (adapted from [KBB16])

Rework results in changes that are committed as a response commit. The commit must be pushed to the remote repository in order to automatically update the merge request with the new changes. Typically, the tool used to conduct the review is integrated with the version control system, so that the commit automatically triggers the update, which alerts the reviewers to return back to the merge request and conduct

the examination activity. This examination review cycle is repeated until the request is approved, at which point the review enters the last step in the workflow.

The work that remains for the **integration** activity, is for the developer to merge the changes into the development branch, as shown in Figure 5.8. Possible impediments that could arise from the merge attempt are merge conflicts. They are present when the source branch has diverged from the destination branch and the two contain different changes in same areas. To resolve the conflict, the developer can either use tools designed for this purpose, or, in the worst case, do it manually.

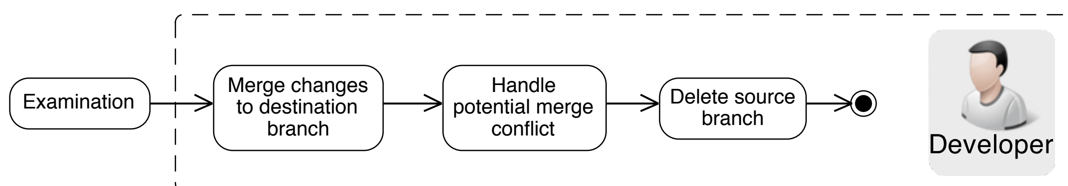


Figure 5.8: Workflow for the integration activity in branch based code reviews (adapted from [KBB16])

The actual resolution involves choosing which version of the changes to keep: the developer's changes, the ones in the destination branch or a combination thereof. If the merge conflict involves large changes to the codebase, the reviewer should examine the code again. Once all conflicts are resolved, the code can be successfully merged into the development branch. The developer can then delete the source branch to clean up the repository, which marks the end of the review workflow.

5.2 Related Work in the Area of Code Reviews

Our results regarding review motivation differ from publications such as [BB13] where finding defects ranks first. However, [BB13] and [RB13] come to the same conclusion that improving readability and maintainability counts for the majority of the feedback in reviews. The studies also confirm our findings that reviewing increases exposure and enables developers to learn more about the system [BB13, RB13]. There is also agreement that reviewing enables teaching novice developers about quality and best practices [BB13]. Similarly, the interviews highlighted that the mere knowledge of code being reviewed and criticized leads to developers paying extra attention to quality and therefore writing better code. Developers reported a remaining need for direct communicate during reviews [BB13].

Developers need incentives for reviews, otherwise they do not like to spend time on reviewing code. Team members who develop many features and fix a lot of bugs are seen as heroes, reviewers do not get noticed so much [BB13]. Teams should

be able to adapt the process to their own needs, e.g. to allow certain changes to happen on the development branch or time critical bugfixes not to be reviewed. In contrast to formal reviews, modern code review approaches involve less formal practices [CBDT06, RCP+12]. Informal practices enable teams to adapt code reviews to their needs and to switch to other forms such as over the shoulder reviews or pair programming.

In open source communities such as GitHub, repository owners manage incoming code contributions using merge requests[2]. Developers without write access fork the repository and implement their contribution in their forked repository. If they want to merge back their contribution, they create a merge request from their forked repository to the original repository, which includes their changes. The owner of the source repository can review the changes and ask for improvements before accepting the change. Publications show that merge requests are an important part of the social coding community in GitHub and improve transparency, learning and collaboration in open software repositories [DSTH12].

[TDH14] presents a study on open source contribution in GitHub that evaluates merge requests, which are the primary method for contributing code. They analyzed the association of technical and social measures with the likelihood of acceptance. They found that repository owners use the information about the technical contribution and the personal connection to the submitter when reviewing the request. In some cases, multiple rounds of reviewing and comments were necessary to establish a shared understanding. This is in line with [MDH13] who found that uncertain merge requests need negotiation and explanation. Merge requests with many comments tend to signal controversy and were less likely accepted by repository owners [TDH14, DSTH12] which is different to Rugby's approach where many comments lead to more improvements of the quality. Popular projects were more conservative in accepting merge requests because it poses a higher risk for the users of the source code if defects get through [TDH14].

[2]GitHub calls merge requests pull requests: https://help.github.com/articles/using-pull-requests

5.3 Release Management Workflow

Figure 5.9 shows Rugby's release model which includes a *Version Control Server* with the same concept as modeled in Figure 5.2 and a *Continuous Integration Server* that has *Build Plan*s that consist of *Build Stage*(s) which can depend on each other and can further be decomposed into *Build Job*s and *Build Task*s. A build plan usually consists of three stages to build a pipeline: *Integration Stage* automatically produces a *Build* for a given *Branch*. The build is tested in the *Test Stage* that produces *Test Result*s. After passing this stage, there is a manual *Delivery Stage* triggered by the *Release Manager*. The delivery stage turns the build into a *Release* with *Release Notes* which can be edited by the *Release Manager*.

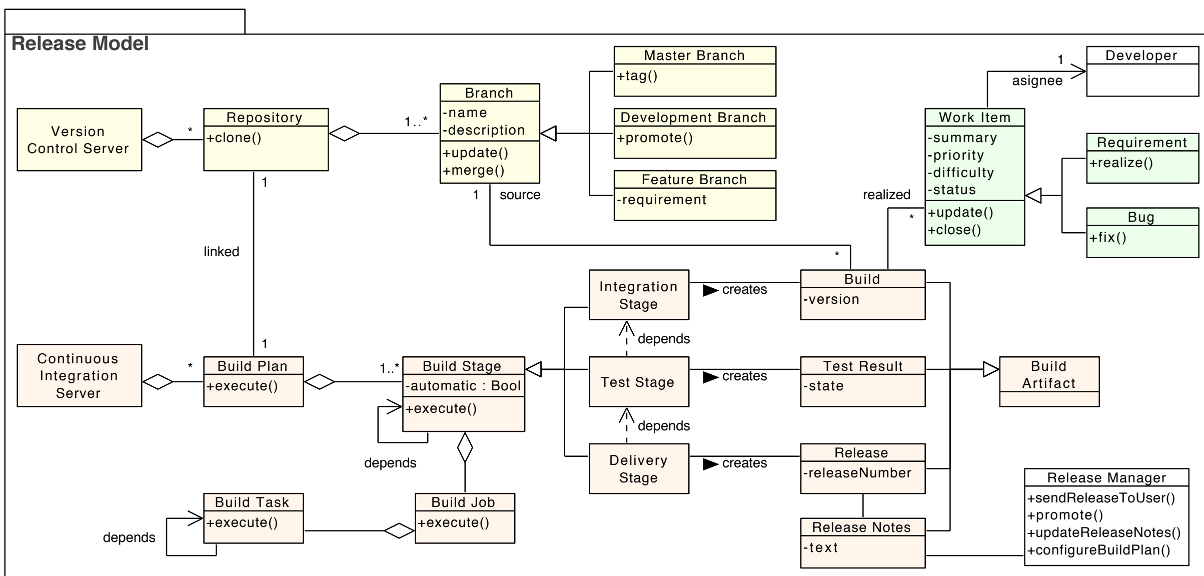


Figure 5.9: Rugby's release model: concepts of the release workflow and their relations in a UML class diagram - colors highlight related concepts

Rugby uses Humble's deployment process as base for its integrated release management workflow shown in Figure 5.10. The release management workflow starts each time a developer commits source code to the version control server, leading to a new build on the continuous integration server. If the build was successful and if all test stages passed, the release manager uploads it to the delivery server which then notifies users about a new release. The release manager can choose to send the release to all users or only to a certain group of users, e.g. A/B testers of a specific feature. He manages users and groups in the delivery server and promotes releases.

Each release includes release notes, which are obtained automatically by the continuous integration server using information of the issue tracker, such as closed backlog

items. Optionally the messages in the commit history in the source code repository can also be used as information source for the release notes, however commit messages might be very technical and not suitable, because an end user might not understand the message. The release manager can edit the automatically obtained release notes in a manual release step if necessary. This semi automatic approach for release notes saves time, allows the release manager to describe features and resolved bugs in terms that users can understand and the release manager can insert specific questions for feedback.

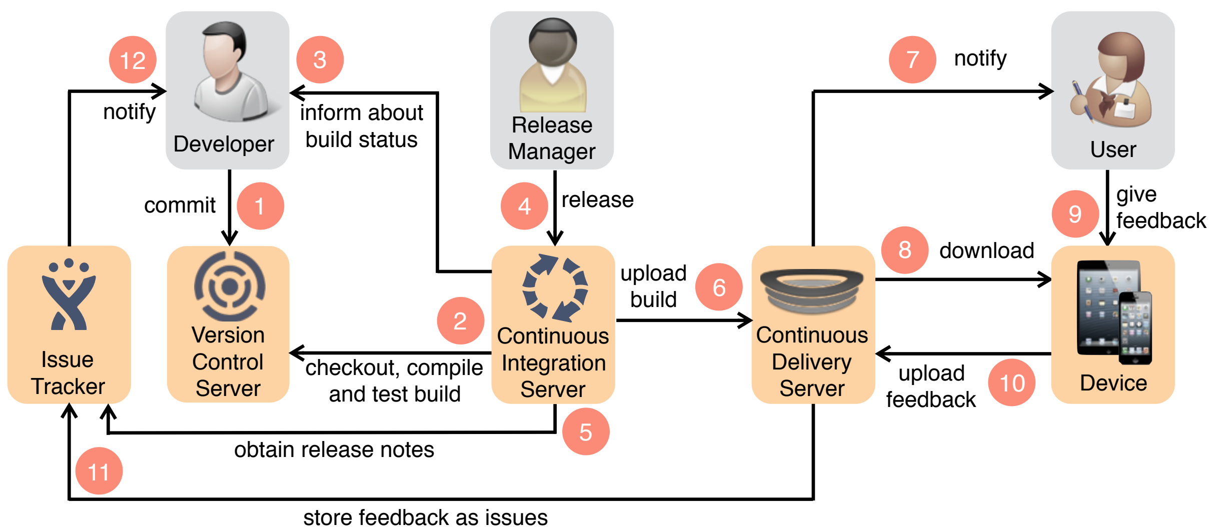


Figure 5.10: High level release management workflow with roles, services and transitions (adapted from [KA14])

The user can download the release and recognize easily, which features and bugs were resolved in the release. He can use an embedded mechanism to give feedback in a structured way. This feedback is collected on the delivery server and forwarded to the issue tracker which notifies the developer about new feedback. The workflow in Figure 5.10 only shows a limited amount of interactions of the developer with the version control server.

In fact a developer has more possibilities than just committing source code. He can create branches to separate the work on a feature basis and merge these branches. Rugby's deployment process can be configured to automatically build and test all types of branches. Continuous delivery combined with branch management helps developers to automatically check if new features pass all tests and can be delivered to the target environment. It also enables developers to let users or customers validate the requirements of a feature, by producing a release in form of an executable prototype as communication model and sending it to the user or customer (event based).

While event based releases are a great opportunity to obtain early feedback, they do not come for free and involve certain cost factors. The team might need time for the decision to create a release and send a request to the user. There might also be effort for deciding about included features and for creating the release, for contacting users and requesting their feedback. There problem could occur that users get too many releases and do not have time to respond to them quickly so the team has to wait for feedback.

While Rugby's release management workflow automates the creation as well as the transmission of the release to the user, it cannot automate the decision when an event based release is appropriate or not. Experience shows that self organizing teams appreciate the possibility and usually find a good balance between too less and too many event based releases so that the described additional time effort can be neglected. Rugby uses executable prototypes to communicate with the user during the whole development process [BKW12]. Rugby allows developers to create releases from any branch. Figure 5.11 shows four different use cases of releases in Rugby.

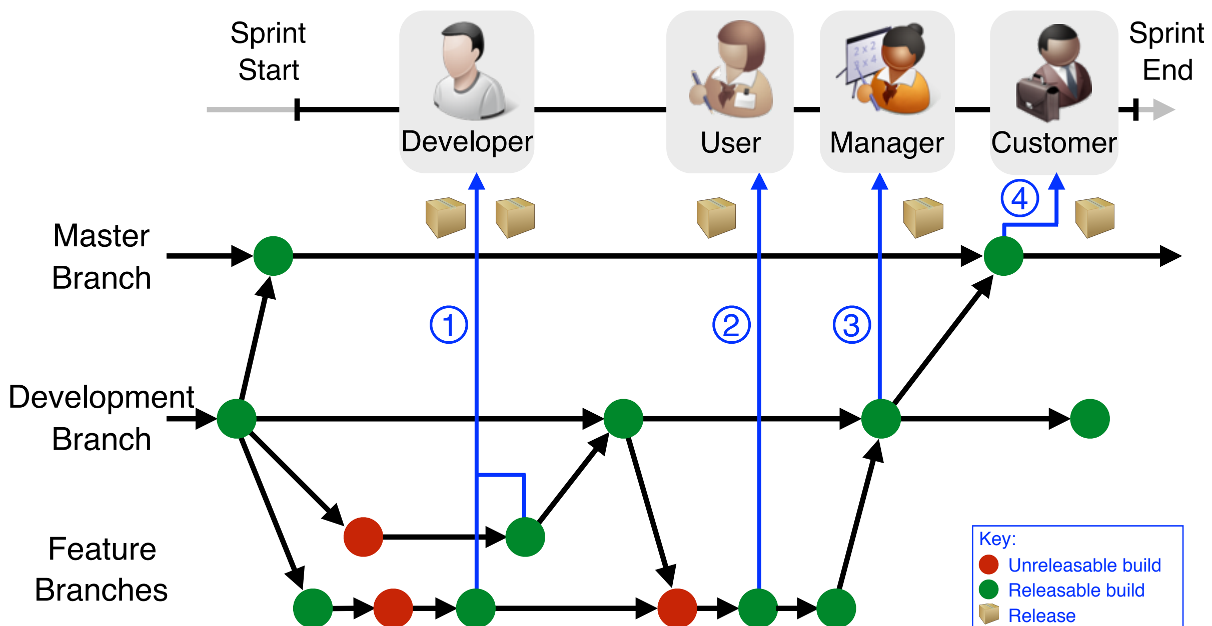


Figure 5.11: Event based delivery in the context of the branching model with four examples for releases (adapted from [KA14])

Releases from feature branches can be used in meetings to demonstrate the development status to all other team members (1). This improves the quality of the communication in the team meeting and shortens the time that is required to explain specific implementation details. Releases from feature branches can also be used to obtain feedback from users to see whether a feature is usable and satisfies all user wishes

(2). In both cases, the release notes contain information about the progress of the corresponding backlog item that is realized in the feature branch, such as a list of open and closed tasks for the backlog item. The developer can insert a clarification question into the release notes or point out on which aspects of the backlog item he needs feedback. This helps the other team members and users to quickly identify the changes of the release with respect to the previous release they received.

Releases of the development branch can be used for the status in management meetings (3). The development branch only contains finished backlog items. Therefore the release notes contain an overview of the realized backlog items, where the ones are highlighted that were not yet realized in the previous release from the development branch. The release notes can also contain the list of backlog items, that are not finished yet, so that a manager can recognize on which items the development is working in the current sprint. Such releases can improve the coordination across teams in project based organizations because the current implementation status of one team is always visible in form of executable prototypes.

Builds from the master branch are used as time based releases in the way Scrum uses product increments at the end of the Sprint (4). This means the master branch always contains the latest potentially shippable product increment. The release management workflow produces releases automatically when the team merges the finished features into the master branch before the sprint review meeting and the team does not need to compile and build the release on their own computer. In this cases, the release notes highlight the list of changes since the last release from the master branch, but also contain older changes so that customers recognize what they can expect in the release and on which items they should focus in the sprint review meeting.

Developers and managers can also select a releasable build and deliver it to their own device for demonstration. The ability to use branching increases the flexibility for the developers, because they now have the possibility to create internal releases to test the software on their own devices and external releases just for specific features. Managers can use a development build with no additional costs to discuss the progress and current issues with other managers, using the same deployment process and amount of automation.

5.4 Related Work in the Area of Release Management

Humble describes release anti patterns as well as benefits and principles of continuous delivery [HF10]. He then illustrates configuration management and continuous

integration as the prerequisites of continuous delivery and presents his model of the build pipeline. He also introduces a management model and a maturity model for continuous delivery and describes the relation to risk management. While Rugby's release management workflow is based on Humble's build model, Rugby includes a mechanism for event based releases.

Rugby also describes organizational aspects such as how to handle feedback that is obtained from releases. All these elements cannot be found in [HF10]. Another related topic in this area is DevOps [Hum11], a concern for larger applications with multiple operators that care about topics such as performance and scalability. Rugby focuses on the shared understanding between developers, customers and users and was introduced and evaluated in smaller, innovative mobile app development projects without the need for operations.

Michlmayr and his colleagues analyzed the release strategy in large open source projects and discuss why and how these projects should adopt time based releases [MFS15]. They found that the introduction of a time based release strategy with regular releases offers several benefits in contrast to non regular feature based release strategy, such as improved quality and predictability of releases However, they did not analyze the impact of event based releases.

Wright describes time based releases and feature based releases in the context of the release engineering process [Wri12]. Rugby also supports time based releases and additionally supports event based releases that can include unfinished features. Rugby's approach for obtaining release notes is similar to the one described by Morena and his colleagues [MBDP+14]. It also uses information of the issue tracker (and optionally information of the commit history in the source code repository) to create information about the changes of the release. The difference is, that Rugby supports different release notes depending on the branch the release is started from (compare Figure 5.11.)

Marschall describes the transformation of a six month release cycle to continuous flow with small releases in a company [Mar07]. His approach is similar to Rugby's release management workflow. He uses test cases as quality gate to determine whether the release can be delivered or not. Rugby also uses test cases as quality gate, but introduces code reviews as additional quality control before time based releases are created.

Feitelson describes how Facebook applied continuous deployment including a combination of review management using a commit based code review workflow and release management using a deployment model with multiple control steps [FFB13]. Instead of using feature branches to separate the development, they apply feature tog-

gles, i.e. small switches that can completely activate or deactivate a feature during runtime. Even if the feature is not realized, its code will be shipped and can be activated for testing purposes for only a small user base of alpha testers.

The IT Infrastructure Library (ITIL) is a collection of predefined and standardized processes, functions and roles that are typically used in medium sized or large companies for IT service management [Kö06]. ITIL defines service transitions which relate to the delivery of services required by a business into operational use. One service transition is release and deployment management, which is used by the software migration team for automated distribution of software and hardware. Typical goals of release management in ITIL include planning the rollout of software and designing and implementing procedures for the distribution and installation of changes to IT systems.

In addition, ITIL release and deployment management is responsible to control how a release is tested and deployed into the production environment. The main goal is to ensure that the integrity of the production environment is protected and that only tested components can be deployed. In this sense, ITIL can be seen as a quality gate to production. While Rugby promotes regular event based releases to test software in the target environment, release managers have to be careful if they release unfinished or untested software into the production environment. Rugby recommends to test software in test environments.

5.5 Feedback Management Workflow

Figure 5.12 shows Rugby's feedback model with *Work Item* and *Feedback* as subclasses of *Issue*, both can include *Comment*s by *User*s and *Developer*s. Certain feedback types can be converted into work item sub types: *Bug Report* into *Bug*, *Design Request* into *Design Improvement* and *Feature Request* into *Requirement*. Additionally, feedback includes *Usage Context* that consists of media *Attachment*(s), *Application Screen*(s) and other information. A feedback belongs to one *Release* that realized certain issues and was created in the release management workflow.

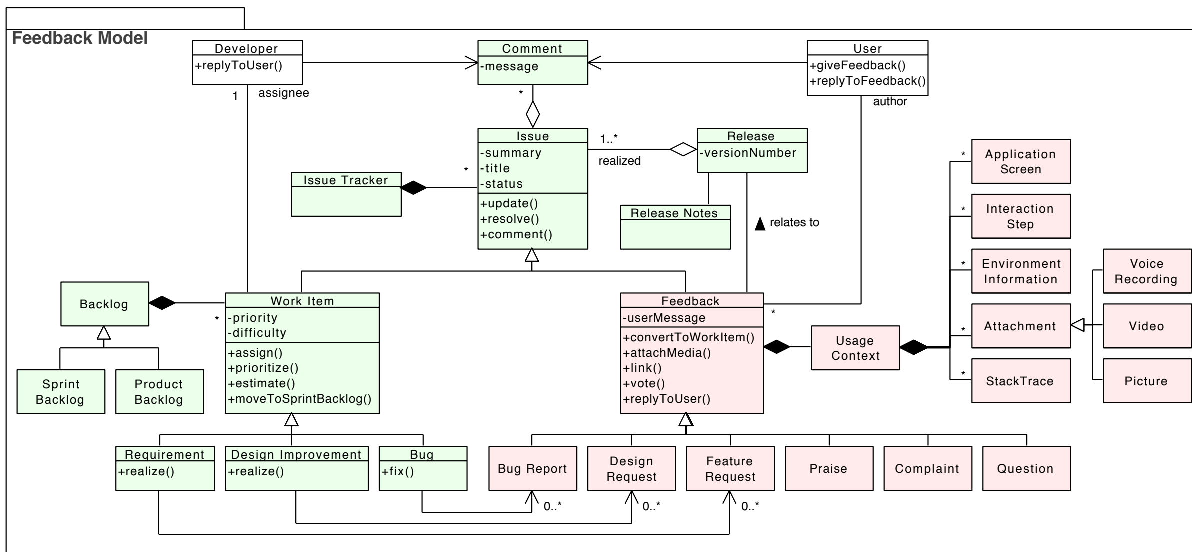


Figure 5.12: Rugby's feedback model shows concepts of the feedback workflow and their relations in a UML class diagram - colors highlight related concepts

While Rugby expects developers to deliver at least one time based release at the end of each sprint, it also allows them to release their software event based, whenever they want to obtain feedback. In this section we first describe a semi automated feedback management workflow and then discuss legal issues, in particular concerns with respect to data collection and live experimentation. User feedback is an important source of elements for the backlog such as bug reports, design requests and feature requests. Instead of eliciting requirements upfront, users try the product increment and provide valuable feedback for the further development using different feedback channels. The level of automation depends on the feedback channel. Crash reports and instrumentation results such as usage statistics are collected automatically with Rugby's feedback framework that is integrated into the delivered application.

Developers can choose to collect heuristics about usability issues in mobile applications, e.g. low discoverability of user interface elements, and measure user interac-

tion [HR00]. The framework measures the usage context (e.g. availability of network connections, location of the user, stack trace of an exception, version of the software, etc.), uploads the results back to the delivery server and attaches the information into the feedback report. Crashes automatically lead to bug reports that are converted to bugs in the issue tracker.

When using the built in feedback mechanism, the usage context (e.g. the currently active view and the current state of the application) is attached to the feedback. The integrated framework suggests the user existing feedback reports to prevent duplicates and allows users to vote or comment [Pag13]. For other feedback channels such as emails, phone calls and meetings, developers still have to insert and categorize feedback manually into the issue tracker. In such situations, sentinel analysis tools can help developers in the categorization and prioritization [GM14].

Figure 5.13 shows Rugby's semi automated feedback workflow and visualizes how developers manage feedback. Depending on the feedback type, developers initiate different workflows to handle it: feature requests in the analysis workflow, design requests in the design workflow and bug reports in the implementation workflow. During Sprint 0, the client receives an empty release, labeled R0 in Figure 5.13, to ensure early in the project that release and feedback management workflows are set up.

R0 contains a feedback button that allows the user to send an empty feedback, labeled F0, within the application back to the developers. The initial release takes longer (indicated by a longer horizontal bar in Figure 5.13), because the release manager needs to setup build plans in the continuous integration server and the mechanism to deliver the application to the user. After the setup is completed, subsequent releases can be created easily and take less time[3].

The first release R1 includes a realized scenario (the team went through all development workflows: analysis, design and implementation; some of the work was done in parallel) and leads to two feedback items F1 and F2. F1 is automatically categorized as bug report and converted to a bug in the issue tracker. The responsible developer decides to fix this bug in the current sprint, so he moves it to the sprint backlog, corrects the bug, commits his changes and releases R2 that includes the bugfix. R2 automatically includes audience specific release notes [KKB16] about the resolved bug, so the user can directly see that the developer was able to resolve it.

While further using R1, the user detects a user interface design problem and produces another feedback report F2. This is categorized as a design request and converted to a design improvement in the issue tracker. While it is additional work, the

[3]Usually the release only takes about one to five minutes depending on the size of the software, after the source code was automatically built in the release management workflow.

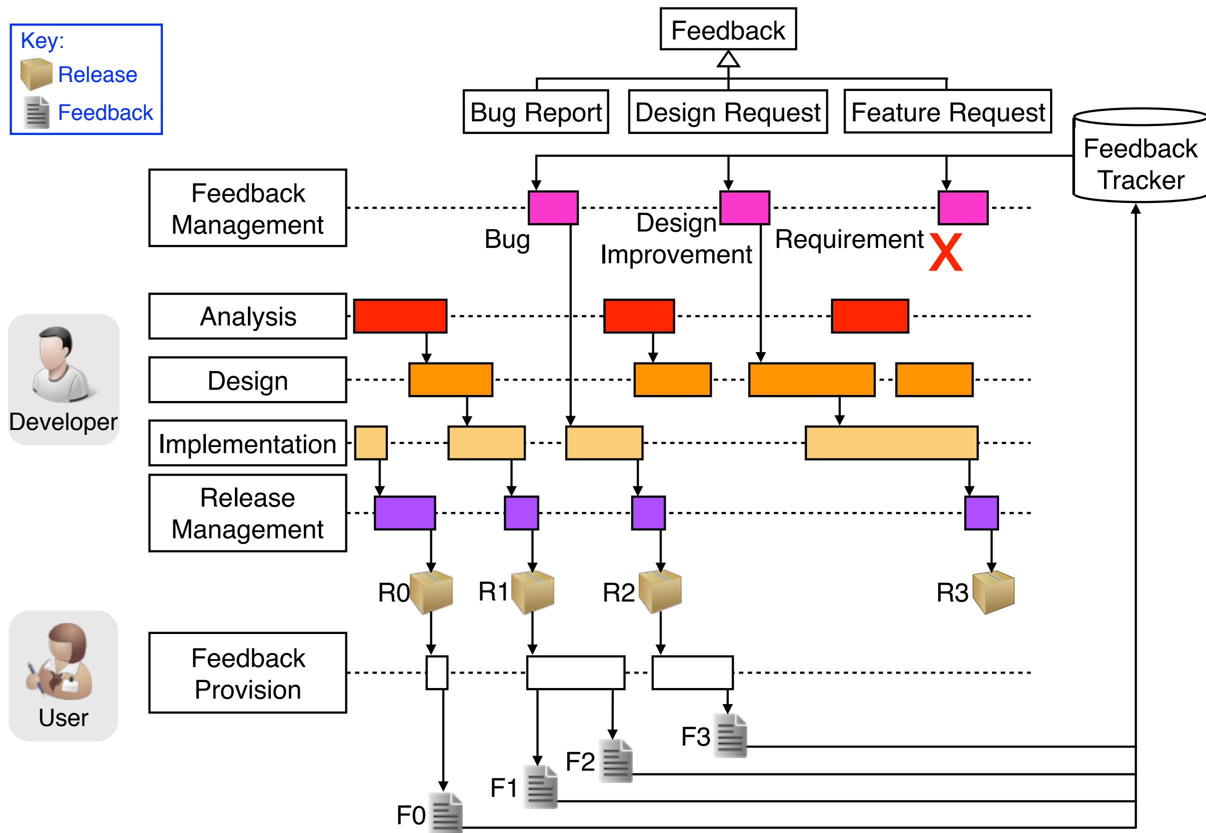


Figure 5.13: Rugby's feedback management workflow (adapted from [KABW14]) shows that feedback requests by users are handled differently depending on their type - the colors are corresponding to the workflows in Figure 5.1

team decides to implement the design improvement within the same sprint because it is related to the already realized scenario. After switching to R2, the user recognizes that the bug was fixed that he reported. He identifies a new requirement and provides another feedback report F3. F3 is a feature request so it will be converted to a requirement. The team decides to put it on the product backlog to review it with the customer in the next sprint planning meeting. Rugby's combined release and feedback management workflows support pulling feedback initiated by developers and pushing feedback (spontaneously) initiated by users.

Figure 5.14 visualizes three feedback situations according to [Sch11]: (1) Developers can pull feedback, e.g. by using Rugby's mechanism to send a release to certain user groups during A/B tests, including release notes with specific questions. (2) Users can spontaneously push feedback using multiple channels (e.g. social media). Then only some feedback reports reach the developer (unfocused) and feedback is lost. (3) A mechanism, e.g. Rugby's integrated feedback mechanism, can focus spontaneous user feedback so that all feedback reports reach the developer and no feedback is lost.

Rugby supports pulling feedback (1) and focusing pushed feedback (3) to prevent that feedback does not reach developers as in the case of unfocused push (2).

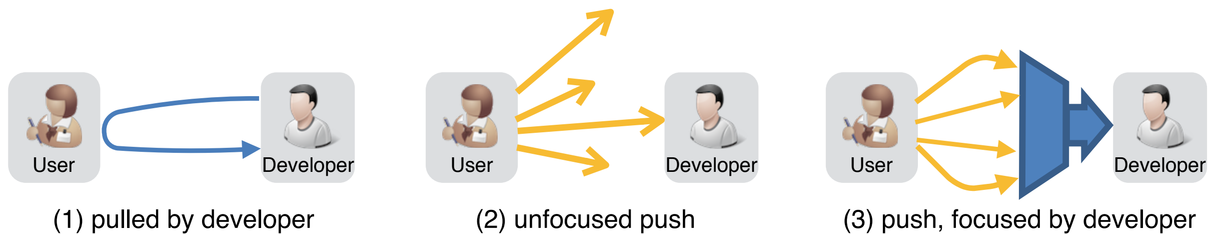


Figure 5.14: Three approaches for feedback provision (adapted from [Sch11])

There are different challenges when continuously analyzing user feedback that might add up to additional effort: quantity of data, missing structure, content and quality and conflicting preferences [Pag11]. Continuously gathering user feedback results in large amounts of unstructured data, complicating manual and automated analysis techniques. Developers have to interpret information provided by users to understand their needs. Gathering user input without mediation results in unpredictable content and quality, which leads to misunderstandings [ZPB+10]. Content and quality of user feedback directly impact how precisely developers can understand users' needs.

User feedback with low quality and inhomogeneous terminology is hard to analyze and might be ignored. Continuously gathered user feedback can lead to conflicts, when different users report contrary opinions [Pag11]. Rugby's feedback system includes a semi automatic categorization mechanism to address these challenges. It automatically collects the relevant context information by instrumenting the application and lets the user annotate it (e.g. by selecting the type of feedback) to precategorize user feedback which improves the structure of feedback reports [SMP+10]. It allows users to see existing feedback so that they do not report duplicated feedback. If the users still provide duplicated feedback, natural language processing approaches as described in [FCC13] could help to detect these duplicates. Users can discuss on existing feedback reports, which can increase the quality of the feedback, and vote on feedback which can solve conflicts.

Though these additional input facilities improve the communication with respect to feedback between developers and users, not all users might use the integrated feedback channel. It is important to make the value of feedback cycles visible to both, developers and users. The four step value chain for software evolution summarizes the values [Sch11]. The user composes and sends feedback, while the developer analyzes the feedback and improves the application. The value chain integrates the

perspectives of senders (users) and their motivation to compose a feedback message with the interests of receivers (developers).

It is necessary to lower the user's effort and to increase his intrinsic desire to provide feedback. On the receiver side, standardized feedback should be filtered and sorted by predefined criteria to lower the developer's effort. This leaves manual analysis time for feedback messages that cannot be interpreted automatically. Rugby uses context information as a clue for reducing effort on both sender and receiver sides [Sch11, SMP+10]. It supports the four focusing steps in the feedback value chain (compose, send, analyze, and improve) [Sch11] by providing a voting and commenting system for existing feedback to lower the user's effort and by providing pre categorization and context information to lower the developer's effort.

5.6 Related Work in the Area of User Feedback

Involving users during any phase of the software lifecycle is a socio technical issue when gathering and analyzing their feedback [Gru91]. Analyzing usage data, uploading usage context and instrumenting the application raises privacy concerns for users. In Rugby, user involvement is supported by technical entities, in particular the software itself as well as the monitored context. These activities involve technical activities of software development, release planning, and deployment.

On the other hand, Rugby aims at establishing a social communication with the end users, asking them to provide information in order to deliver a better service. Dealing with technical aspects has always been part of software development, the social aspects of it have only recently picked up momentum [MP11]. A number of gaps typically complicate the communication between users and developers. These gaps can be of both social, e.g. trust, and technical nature, e.g. communication media.

Important questions answered by socio technical goals include: How can trust between developers and users be established and maintained [TF99]? Which information may be monitored, which feedback is regarded helpful, how much effort are users willing to offer? Tseng and Fogg discuss the interaction and relationship between trust and credibility [TF99]. If developers want to include feedback, they have to take concrete decisions, e.g. which information they want to obtain from users. Rugby aims at maximizing the benefit for both users and developers. For that purpose, the socio technical goals for the project need to be established, explicitly involving both users and developers perspectives.

Rugby's feedback collection mechanism is responsible for obtaining end user feedback in a non intrusive way, to associate it with relevant context information, and to send it to the application developer [GM14]. It uses an application independent framework for mobile applications including specific feedback views, which depend on attached context information. After user feedback is collected, it is enriched with the gathered context information and sent to the continuous delivery server which collects the feedback and forwards it to the issue tracker as shown in Figure 5.10.

Challenges for developers lie in identifying which information to ask from the user, and in processing the monitored context information with respect to relevance and privacy . Rugby's feedback mechanism allows end users to attach rich information such as videos, pictures, and audio recordings. User feedback is linked with the gathered contextual information, to obtain contextualized feedback. Linking the feedback to the issues in the issue trackers and the ability to convert feedback into requirements or bugs provides feedback traceability.

There are many different ways how to obtain feedback after the initial deployment of an application such as the upload into an app store. If an application is already deployed, an integrated feedback framework cannot prevent users from sending feedback over unstructured channels like app store reviews or social media. App stores allow users to submit feedback for downloaded apps in form of star ratings and text reviews [PM13].

Such feedback includes diverse information useful for developers, such as user requirements, ideas for improvements, user sentiments about specific features, and descriptions of experiences with these features [GAB15]. However, for many applications, the amount of reviews is too large to be processed manually and their quality varies largely [GM14]. Star ratings apply to the whole application and it is often hard for developers to analyze the feedback for single features.

Natural language processing techniques allow to automatically identify features in reviews, extract sentiments about a feature and categorize the feedback among certain features. Such approaches can help developers to systematically analyze diverse user feedback and filter irrelevant reviews [GM14]. While Rugby describes how to handle feedback reports, it does not include mechanisms to obtain feedback from external sources such as app stores and natural language processing techniques to automatically analyze this feedback.

Chapter 6

Case Studies

> "To acquire knowledge, one must study; but to acquire wisdom, one must observe."
>
> — Marilyn Vos Savant

In this chapter, we present three case studies in which we used Rugby. Table 6.1 shows an overview of these case studies.

ID	Year(s)	Environment	Projects	Participants
C1	2011 - 2015	6 Capstone Courses	62	579
C2	2015	1 Lecture	57	about 400
C3	2014	8 Industry Projects	8	31

Table 6.1: Overview of the case studies

Section 6.1 describes the first case study C1, a multi customer capstone course with up to 100 participants and up to 11 parallel projects with customers from industry. We used Rugby in six capstone courses with 62 projects and overall 579 participants between 2011 and 2015. We describe interventions – incremental improvements of the learning experience – and the course environment. We also show how the students learned Rugby's workflows to apply them in their projects.

Section 6.2 presents the second case study C2, in which we used Rugby in a university lecture with about 400 registered students. Students learned Rugby's workflows in individual exercises and then applied them in small projects in team based exercises to improve their project management skills.

In Section 6.3, we present the use of Rugby's release and feedback management workflows in eight industry projects in a project based company Capgemini as the third case study C3. We extended Rugby's release and feedback management workflows and customized them for different project sizes.

6.1 Capstone Course

Since 2008, we have regularly conducted a multi customer capstone course in the university which we call “iOS Praktikum”. In this capstone course, student teams develop mobile applications for the iOS platform in innovation projects using the newest technologies for real industry clients such as BMW, T-Systems or Siemens. The iOS applications are embedded in a larger system context, i.e. the apps communicate with backend servers, with smart sensors such as iBeacons, intelligent clothing, wearables like smart watches or micro controllers such as the Raspberry Pi or the Intel Edison.

The idea is to make the project as realistic as possible to increase the learning experience of the students. The students should interact with real data from the clients and therefore sign a non disclosure agreement. If the development pressure is high, the students experience real problems in their team work and in the communication with the clients. The projects focus on the development of innovative applications over a period of one semester, i.e. three months, so the students have a real deadline to finish their application.

Between 2008 and 2014, we conducted the capstone courses once a year in the summer semester (SS). The capstone courses were so successful and the students reported such a great learning experience, that we decided to conduct the capstone course twice a year since 2014. In addition to the summer semester, we also offer the course in the winter semester (WS).

Around two thirds of the developers are master students who already developed object oriented programs before. Their experience ranges between a few months and up to four years of developing. One third of the developers are bachelor students with no or limited development experience. One project team includes one project leader, one student coach and between six and eight student developers[1].

Table 6.2 shows the number of participants in these six courses between 2011 and 2015. It also shows the average team size including student developers, student coaches and project leaders for each course, which ranged between 7.8 in SS 2011 and 10.6 in WS 2014/15.

[1] There were a few exceptions to this distribution: In SS 2011, no student coaches participated in the course. 1 project had five developers. In SS 2012, 1 project had 4 developers, another project had 2 project leaders. In SS 2013, 1 project had 2 project leaders, another project had 13 developers and 2 coaches managing 2 sub projects in the same system context for the same client. In SS 2014, 4 projects had two project leaders who shared the management work. In WS 2014/15, 4 projects had multiple sub projects in the same system context for the same client: 1 project had 2 project leaders, 1 coach and 10 developers. Another project had 3 project leaders and 12 developers. The third team had 2 project leaders, 2 coaches and 12 developers. In SS 2015, 1 project had 2 project leaders.

Year[2]	# Projects	Team size (on average)	# Student developers	# Student coaches[3]	# Project leaders[4]	# Program managers
SS 2011	8	7.8	54	0	8	3
SS 2012	11	8.4	69	11	12	3
SS 2013	10	10.1	79	11	11	3
SS 2014	11	9.5	79	11	14	3
WS 2014/15	11	10.6	90	12	15	4
SS 2015	11	9.4	80	11	12	3
Total	**62**	**9.3**	**451**	**56**	**72**	**19**

Table 6.2: Number of participants in the multi customer project courses between 2011 and 2015. The average team size includes student developers, student coaches and project leaders.

6.1.1 Interventions

In 2010, the process model in the then called iPhone Praktikum was a rather linear one with some iterative elements such as the design review after two thirds of the course. In the beginning of the projects the focus was on requirements elicitation and analysis, in the middle on system and object design that was reviewed in the design review presentations. In the last four weeks the students then focused on the implementation.

Table 6.3 shows the interventions in the iOS Praktikum between 2011 and 2014. In 2011, we introduced Rugby as agile process model that was used from the beginning of the course. The students learned the main concepts of Scrum in a course wide tutorial two weeks after the kickoff. We explained how to apply Rugby as a variation of Scrum, with part time developers and weekly meetings. It was also the first time, that we asked the teams whether they want to use git as distributed version control system or SVN as central version control system. Half of the teams decided to use git and the other half decided to use SVN.

In the next instance of the capstone course in 2012, we integrated the idea of an initial elaboration phase into Rugby's process model and called it Sprint 0. We decided that distributed version control is the new standard and that all teams should learn and use it. In addition, we switched the tools that the students used for the development

[2]More information about the capstone courses can be found on the corresponding websites. SS 2011: http://www1.in.tum.de/ios11, SS 2012: http://www1.in.tum.de/ios12, SS 2013: http://www1.in.tum.de/ios13, SS 2014: http://www1.in.tum.de/ios14, WS 2014/15: http://www1.in.tum.de/ios1415, SS 2015: http://www1.in.tum.de/ios15

[3]In rare cases the same student was coach in two teams. In such cases, we count this participant twice to calculate an appropriate team size.

[4]In rare cases the same person was project leader in two teams. In such cases, we count this participant twice to calculate an appropriate team size.

Year	Interventions
2011	First application of Rugby as agile process model Introduction of distributed version control using git for half of the teams
2012	Adaption of Rugby's process model to include Sprint 0 Introduction of Atlassian JIRA for issue tracking Introduction of Atlassian Confluence for knowledge management and meeting Distributed version control using git for all teams Introduction of release and feedback management with Atlassian Bamboo and HockeyApp Introduction of functional (cross project) teams
2013	Introduction of Atlassian Stash as version control server with web access Usage of a simple branching model Release and feedback management with Atlassian Bamboo and HockeyApp from the beginning
2014	Adaption of the branching model to include merge requests Review management from the beginning: branch based code reviews

Table 6.3: Interventions in the multi customer capstone courses between 2011 and 2014

in 2012. Instead of using the application Unicase[5] for issue and meeting management, we decided to use Atlassian JIRA for issue tracking and Atlassian Confluence for knowledge management and meeting management.

The main advantage was that JIRA and Confluence were easier to use and to maintain, because they were hosted as web services and available in the browser making collaboration easy. Meetings could be created as pages with templates in Confluence and it was possible to reference JIRA issues. We also started to use continuous integration and continuous delivery in 2012 with Bamboo and HockeyApp. Due to problems with the setup of continuous integration for iOS applications[6], the students could only use it about two months after the course has started.

In 2013, we learned from the problems with continuous delivery in 2012, and focused on introducing continuous delivery right from the beginning. We also created dedicated tutorials for release and feedback management to simplify the setup in each team. Another improvement in 2013 was the use of Atlassian Stash as web service for the version control system. Stash simplified the maintenance by allowing an easier setup of git repositories and an easier configuration of access rights. It was now possible for the management and the course organization to see repositories, branches and commits in the browser, to see metrics about the use of version control and to get notifications. We also started to use a simplified version of git flow [Dri10] as branching

[5]Unicase is an Eclipse based application with a unified model for task, meeting and rationale management, which is only available as Desktop client: http://www.unicase.org

[6]The support for building an iOS application with Xcode on the command line was limited in 2012.

model using feature branches for the actual development. Table 6.1 shows an overview of the services and tools in the capstone course since 2013.

Service	Tool	Website with more information
Issue Tracking	Atlassian JIRA	http://www.atlassian.com/jira
Knowledge Management Meeting Management	Atlassian Confluence	http://www.atlassian.com/confluence
Distributed Version Control Code Review	Atlassian Stash	http://www.atlassian.com/stash
Continuous Integration	Atlassian Bamboo	http://www.atlassian.com/bamboo
Continuous Delivery	HockeyApp	http://www.hockeyapp.net

Table 6.4: Overview of the used tools in the capstone course since 2013

In 2014, we adapted the branching model to include branch based code reviews. Developers did not directly merge their feature branches into the development branch when they had realized the feature. Instead, they had to request the merge first. Then, a reviewer, another more experienced student in the team, inspected the code in the browser in Atlassian Stash for flaws, problems and bugs. After reviewing and possibly improving the code in multiple rounds, the request was finally approved and the changes in the features branch were merged into the development branch. This additional step in the branching model allowed for continuous review management from the beginning of the course instead of inspecting the code only once or twice at the end of the project as we did it in previous years.

6.1.2 Course Environment

Figure 6.1 shows an instantiation of Rugby's ecosystem (which is also shown in Figure 4.2) which includes the management roles of the capstone course. The student developer is an instance of Rugby's developer role and the industry client is an instance of Rugby's user role. In addition, project leader and coach represent the management and there is an additional role of the program manager who is comparable to an instructor. Typically three program managers organize the whole capstone course.

Before the course starts, the program managers advertise the course, talks to potential clients, brainstorms with them about possible ideas and their involvement. The clients in our courses come from large companies, small companies with only a few employees, and even include startups. Important milestones and the timeline of the course are shown in Figure 5.1. During the *Kickoff*, the customers present their project ideas. After that, the program managers map students to specific projects using their preferences and balancing constraints. Then, the projects start. The *Design Review*

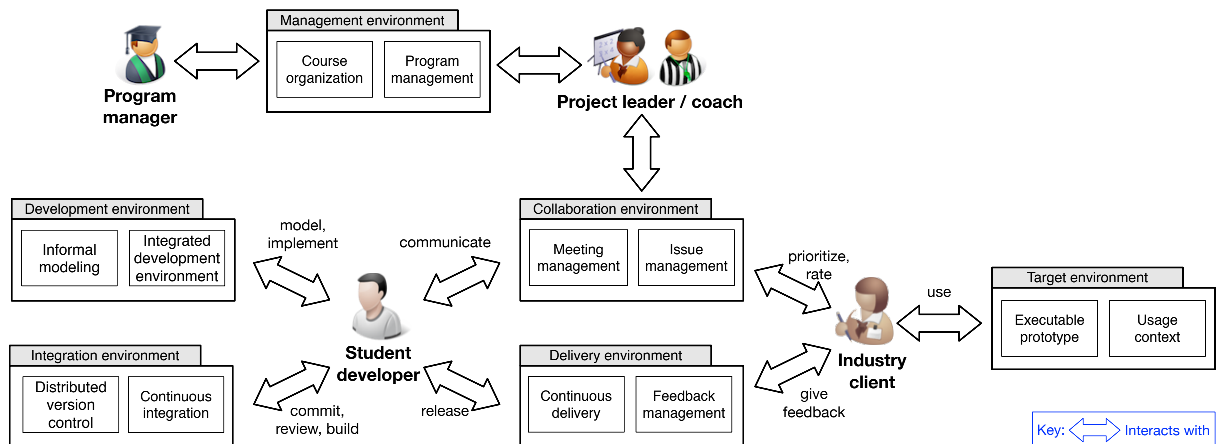


Figure 6.1: Environments of the capstone course including Rugby's ecosystem and a management environment or the general course organization (adapted from [BKA15])

after two thirds of the course is a course wide presentation where all teams show their results of the requirements analysis and system design. During the *Client Acceptance Test (CAT)*, all teams present their results at the end of the semester.

The first phase of the course can be mapped to the inception phase of the Unified Process and lasts until the Kickoff. The program managers provide a template for the *Problem Statement* to the clients who write at least one visionary scenario [BKW12]. The clients are also encouraged to specify an initial *Top Level Design* of the application to be developed so that the students already know whether it is a reengineering project, a greenfield project or an interface engineering project.

The Kickoff is the official start of the course. The clients give a short presentation (about 10 minutes) where they try to convince the students of their project idea. After the kickoff event, the students fill out an online questionnaire, in which they state their preferences for the presented projects as well as their experiences in software development. On the day after the kickoff, the program managers use the results from the questionnaire to assign the students to project teams. This is a semi automated phase that still requires about half a day. Program mamangers try to address the preferences of the students. If all students get their first choice, they know that they stay motivated.

However, there are several constraints to produce balanced teams, e.g. with respect to diversity and gender. On average, about one third of the participating student have a good background knowledge in software development, so that they are considered as experienced developers. Furthermore, about 15 % - 20 % female students apply to the course. Overall there are about 50 % international students.

During the team assignment, the program managers attempt to staff each team with experienced as well as inexperienced students and with a good gender balance.

However, often not all these constraints can be applied simultaneously. There have been kickoff events, where one client attracted all the first choice votes. In other cases, students were assigned into a project that was their fifth or sixth choice. In such situations, it is important that the program managers meet face to face with the affected students. It helps to tell the students that their learning experience is independent from a particular project.

After the team assignment, the program managers set up the team spaces in the collaboration and integration environment and provide meeting agenda templates for the first team meetings. The project leaders invite their team members to the first meeting, in which they explain the basic meeting management concepts and discuss the problem statement. Then, each team starts the initial sprint which we have coined *Sprint 0*. The focus of Sprint 0 is not on development, but on team building exercises. It can be mapped to the elaboration phase of the Unified Process. For example, we use icebreakers that focus on teamwork, in particular team based problem solving, and provide a lot of fun. Example of icebreakers are tricks where the students learn how to rip a phone book in half or the marshmallow challenge [Wuj10]. In addition, each team has to produce a trailer, a short 60 second movie describing the basic idea or vision of the system to be built. The trailer is marketing oriented and helps to bring the client and the team together. In many cases, our clients have used these trailers to market the project within their own organization.

Other team building activities include kart races, paintball games and dinner groups. Such activities help to overcome cultural differences in the team formation phase. Sprint 0 also covers short tutorials about Rugby's workflows to bring all team members up to a shared knowledge level. Our tutorials are based on experiential learning, to establish a culture of continuous improvement and continuous learning within the course [Kol84]. Another activity in Sprint 0, often in the middle as shown in Figure 5.1, is the creation of a first release. Because it is early in the project, students only have to build an "empty release" so that they become familiar with release management techniques, in particular version control, continuous integration and continuous delivery, as well as with feedback management. This empty release R0 can be considered as the executable architecture baseline, because all subsystems need to be included.

After Sprint 0, the teams move on to development sprints which can be mapped to the construction phase of the Unified Process. Each of these sprints usually lasts between two and four weeks[7], depending on the innovation of the project. With students working on different schedules, daily meetings are hard to schedule, therefore the

[7]Explorative projects have shorter sprints as requirements change more often and more feedback is required. Projects with mature requirements usually have longer sprints.

teams conduct weekly meetings (in contrast to Scrum). We consider our students as part timers, because they are taking other classes and exams throughout the course. Part time developers are becoming common in agile industry projects [Fow01]. In addition to the weekly face to face meeting, the students use asynchronous communication mechanisms such as chat as well as audio and video conferences.

In the design review event after two thirds of the course, all teams present their understanding of the problem, show the trailer, the requirements and one or two visionary scenarios usually in form of a demo, as well as the status of the project. The demo can still include workarounds and mocks. However, we require from the students to report which parts of the demo are already implemented, which of them are only unit tested and which of them are bridged by a “narrator”. The work load increases significantly before the design review. The course wide presentation motivates the students because they have to present their work to all other teams in the presence of all clients. After the presentations, the teams get feedback from their clients, as well as from other teams and the program managers.

At the end of the course (after three months), the students present the requirements and the architecture of the system combined with another demo in the course wide client acceptance test. The demo is based on the demo scenario, a refined version of one or more of the visionary scenarios from the problem statement. It should not include workarounds and mocks any more. The CAT (Client Acceptance Test) is filmed and we also provide a live stream into the internet so parents, friends, and others can watch the event online. This allows clients, who cannot be physically present, to see the presentations from a remote location.

The transition phase starts after the client acceptance test and depends on the intentions of the client. Possibilities include a project extension immediately afterwards, usually with some of the students of the team and a productization project where the prototype produced by the students is turned into a product. Often, the results of the project are used for another project course in the following semester.

Figure 6.2 shows the project based organization of Rugby. Each development team is shown as vertical column, consisting of up to eight developers, a coach and a project leader. Each team is self organizing and therefore responsible for all aspects of development and delivery of the software. The project leader and the coach fulfill a role similar to the *Scrum Master*, but in a master apprentice relationship. While the project leader is already experienced with project management, the coach is a student who took a project course in a previous year (similar to the organization described by [JBF13]). This ensures the coach is familiar with the infrastructure and the organizational aspects of our ecosystem.

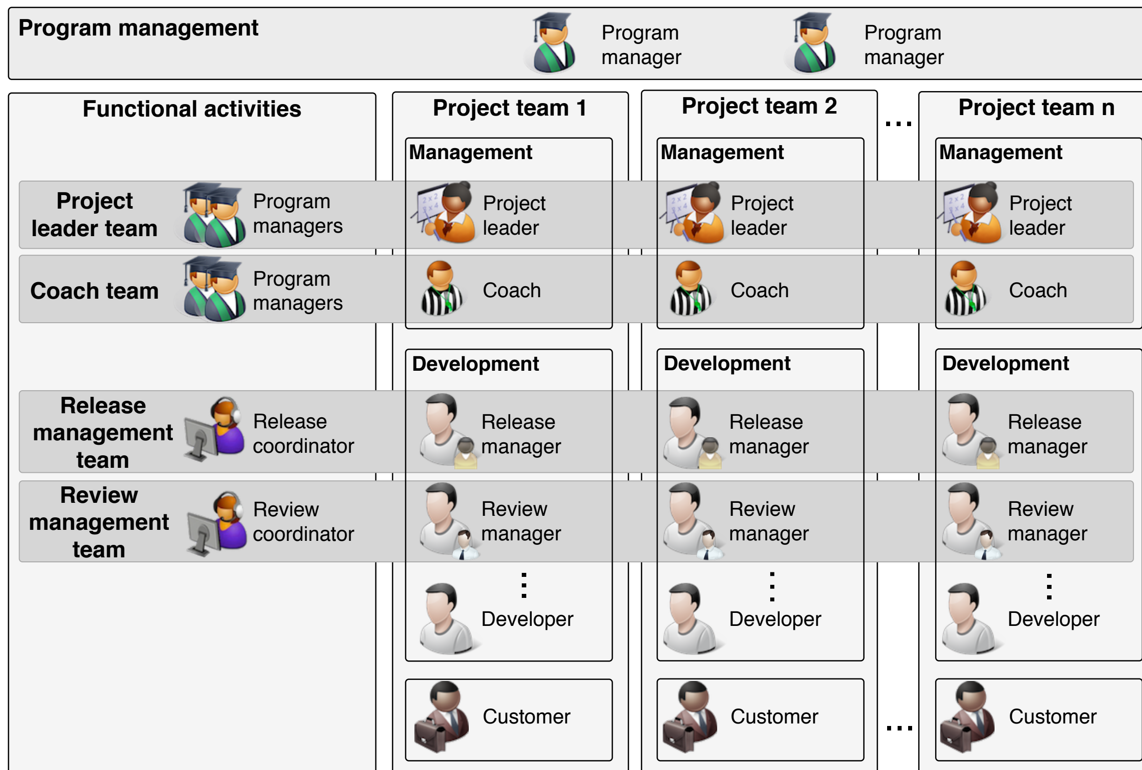


Figure 6.2: Organizational chart showing the project based organization of the capstone course with project teams and functional teams (adapted from [BKA15])

One task of the coach is to organize the first team meeting and to ensure that the team organizes all following team meetings in a structured way. In the first team meeting, the coach takes the role of the primary facilitator and introduces the other two important roles in a meeting, the minute taker and the timekeeper [BD09]. In the following meetings, we require that these roles are rotated between the students in each team, so that everybody learns how to delegate and how to fulfill responsibilities. Börstler also used rotation in his early courses, but in a different way [Bö01].

During the project, the coaches learn essential management skills by observing the behavior and actions taken by the project leader. Another task of the coach is the communication of problems to the project leader and to the program management (see Figure 6.2). The client has a similar role as the Product Owner. If the client is not available due to time reasons or a large physical distance, the project leader takes the role of a proxy client [BD09]. Several functional (cross project) teams are set up to bring software engineering expertise into the development teams (they are represented as horizontal rows in Figure 6.2).

The release management team consists of one student from each project team. It is responsible for release and feedback management issues with respect to version

control, continuous integration and continuous delivery. We also set up cross project teams to address architecture issues and ensure code quality (review team). Membership in the cross project teams is voluntary, because it requires the members to be part of two teams, their project team (which focuses on development) and the functional team (which focuses on a particular workflow). Usually, we ask the most ambitious students to participate in one of these cross project teams. The cross project teams meet regularly to build up and share their knowledge and understanding of tools and workflows. In addition, they often help to resolve conflicts among teams.

Disagreements within team members are taught to be normal, especially during system design, when architectural alternatives are discussed and need to be reviewed. We teach the students that they provide valuable opportunities to develop better teamwork skills and better end products [JJS91]. To help students handle disagreements and tensions in a productive manner, we provide them with syntactical phrases they can use to keep a meeting on time, voice objections constructively, express preferences for certain proposals and reinforce listening skills. Most of these examples are taken from Doyle's book [DS76]. We teach them about the Harward conflict resolution model to resolve conflicts by depersonalization [FUP11]. Actual examples from the team meetings that caused tension (e.g., a domineering personality, a slacker, cultural differences in communication style, heated discussions about alternative proposals) are used by the program managers to demonstrate, which techniques the students could have used to get consensus and arrive at a resolution. This is usually done during the meeting critique at the end of each meeting.

The collaboration environment supports synchronous as well as asynchronous communication, in particular it includes meeting management and issue management. We use a defined structure for meeting agendas and protocols, adapted from [BD09]. The main purpose of our meetings is to have everyone taking away action items and meeting minutes. Meeting skills are required for all software engineers in order to meet efficiently and to avoid information loss. However, meeting procedures and meeting skills are usually not included in standard software engineering curricula. How to make meetings work [DS76] and Mining Group Gold [Kay90] (from which we derived the agenda and protocol templates) describe many useful procedures and heuristics for conducting efficient meetings.

We use a defined structure for meeting agendas and protocols, adapted from [BD09]. During the weekly face to face meetings, the students communicate their status, identify impediments and conflicts. Conflicts and open issues are resolved in the discussion part of the meeting, leading to action items where the students promise to finish identified tasks until the next meeting. In addition to the planned weekly meeting with a fixed

time slot, the teams also agree on working meetings where they solve tasks in smaller groups. To further synchronize their work, they use chat rooms and mailing lists as well as tools like Skype to setup virtual meetings.

We use an issue tracker to allow students to structure their work. They can store product backlog items as well as other tasks. In Scrum, task management is usually done on a physical taskboard, e.g. a whiteboard, because developers work full time in the same room. In Rugby, we use a digital taskboard that is integrated into the issue tracker to synchronize the communication between developers and managers and to allow everyone to know who is currently working on which task.

Our issue tracker supports sprint planning and task estimation. During sprint planning, developer assume the responsibility for a specific sprint backlog item and assign it to themselves. Then they create sub tasks that involve other developers of the teams in the realization of the backlog item. With the help of the digital taskboard, the coach and the project leader can check that each developer has enough tasks to work on in order to balance the task allocation in the team; often the more motivated students assign themselves too many tasks.

During development sprints, project leader and coach track the progress with digital burn down charts. We teach the students techniques such as planning poker [Hau06] or the team estimation game [Joh12] for task estimation. Because planning can take a lot of time, we limit the time for planning, especially in the initial sprints, when the students are not yet experienced with these techniques. Our meeting templates provide the capability for linking issues from the issue tracker directly into our knowledge management system, so that tasks and promises from the current meeting can be tracked and included in the agenda for the next meeting.

6.1.3 Teaching Approach

In this subsection, we explain three ways to teach Rugby's workflows to students in the multi customer capstone course:

1. **Introduction course** before the capstone course starts
2. **Functional teams** (cross project) during the capstone course
3. **Interactive tutorials** in course wide meetings during the capstone course

Introduction course

Before the kickoff of the capstone course, all students attend an introduction course, in which we teach the programming language, common design patterns and important

frameworks of the development platform. The example code of the introduction course is often reused by the students in their projects. Therefore, we check all materials against a comprehensive set of coding guidelines. During the introduction course, students learn the branching model and use these coding guidelines. They develop the solution to programming exercises in feature branches in their own repository and open a merge request to submit their solution.

Tutors review and approve the merge request, if the solution is correct. Otherwise, if anomalies or errors occur, they write comments to ask for changes. If the tutors finally accept the solution, the student merges the changes. With this method the students apply the code review workflow more than ten times during the introduction course and incorporate it in an organic manner. In addition, the students have to setup a build plan on the continuous integration server and write test cases during the introduction courses to get familiar with the concepts of release management and regression testing.

Functional teams

Two typical functional (cross project) teams are shown in Figure 6.2: review management and release management. Each cross project team is led by a coordinator (e.g. a teaching assistant), who is experienced with the topic. The review coordinator is e.g. an experienced programmer who also has good communication skills. One member of each project team is part of a functional team and has the role of the review manager (respectively the release manager). The review manager is responsible for all activities to achieve high design and code quality. The release manager is responsible for all activities to continuously deliver software and to obtain user feedback.

The first cross project meetings are early after the course started and the infrastructure has been setup. After becoming acquainted, the review coordinator explains goals and responsibilities of the review team. He gives a tutorial about branch management and the review workflow using an example project and shows all steps from the developer and reviewer perspectives. The goal is that each team tries out the workflow in Sprint 0 and creates at least one empty review so that the team knows how to apply review management. The review coordinator is the main contact person for the review managers. He needs technical expertise to understand the process and common errors, e.g. how to fix a merge conflict. He also needs skills to communicate with the team of review managers and to track the status of all projects.

The review coordinator introduces coding guidelines and asks the review managers to discuss and agree upon them with their development team. He takes care that all

review managers understand and apply the workflow in their teams. The review team meets biweekly to share knowledge about tools and workflows, to synchronize their understanding and to discuss and resolve potential issues with workflows and tools.

The coordinator uses different techniques to further build knowledge in the team. He assigns small challenges to review managers such as to present best practices, code smells or antipatterns that were reviewed within their team. Another task is to describe how the development team actually uses the workflow and why they might differ from the presented one. This facilitates that review managers take responsibility for their role and internalize the knowledge required for peer learning with the rest of the team. The review coordinator regularly checks whether the review managers fulfill the mentioned tasks by talking to coaches and project leaders.

In the release management team, the coordinator uses a similar approach to introduce the topics and to control that each release manager applies the workflows as needed. He makes sure that each release manager sets up and configures continuous delivery as needed, including build plans for the actual application and for backend services. Each release manager has to create an empty release in Sprint 0 that includes all subsystems of the developed software with one class that e.g. prints a message.

The release manager has to communicate with the rest of the team so that the top level design of the project is integrated in this executable architecture baseline. Some ambitious teams already include first user interface elements into the empty release to get feedback about the navigation concept of the application. In addition, the release coordinator monitors all build plans and helps the release managers if build problems occur in their project team.

The release manager introduces the idea of a wallboard that shows the current integration and delivery state of all branches within a project or all projects within the whole capstone course. An example of such a wallboard is shown in Figure 6.3. Working build plans are visualized in green and broken build plans are visualized in red. We observed that the wallboard visualization motivates the students to always have a potentially releasable version of their software.

Interactive tutorials

The capstone course includes a weekly course wide meeting for milestone events like kickoff, design review and client acceptance test, and is also used in between to introduce workflows and best practices, and to reflect over the current status. The program managers use interactive tutorials to incrementally introduce students to the methods and workflows needed for the project. During these tutorials, the students use the ex-

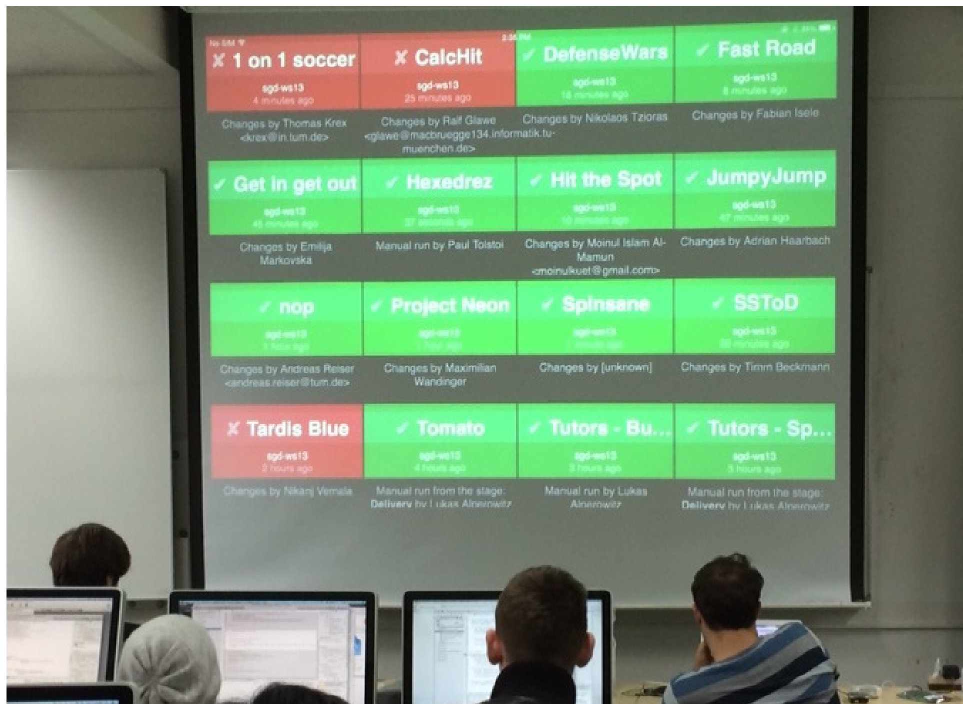


Figure 6.3: Wallboard showing whether the current version of the software is releasable or not

isting infrastructure and experience the workflows hands on before they apply them in their own project team. Table 6.5 shows the content and schedule of these course wide meetings in a three month project course.

The interactive tutorials in the beginning focus on agile methods, meeting management and user interface prototyping to get the projects started and to prepare the students for the deliverables in Sprint 0, such as the initial version of the product backlog. The review coordinator uses one of these meetings to hold an interactive tutorial about review management and the code review workflow. The release coordinator also uses on these meetings for an interactive tutorial about release and feedback management. The main goal of the tutorials is to create a common understanding with all students.

In the class about review management, the review coordinator explains version control, branch management and discusses important best practices such as having small commits, using meaningful commit messages or only committing if the code compiles without errors. Then, he shows how to use merge requests to conduct code reviews in an asynchronous way and introduces best practices such as short branch lifetimes and how to handle merge conflicts. He also introduces coding guidelines, typical code smells and antipatterns as well as examples for refactored solutions. He intermixes theory with practical exercises where the students build pairs to try out the concepts: one student is the developer and another one is the reviewer. They apply the whole

Week	Topics
1	**Kickoff**
2	Icebreaker
3	Agile Methods; Meeting Management
4	User Interface Design [ND86, NH93] and Prototyping [STG03, RSI96]; Trailer and Software Cinema [COB06]
5	Branch and Merge Management; Review Management; Quality Management;
6	Release and Feedback Management; Continuous Delivery
7	From informal to semi formal Modeling [BKW12]
8	Scenario based Design; Software Theater and Demo Engineering [XKB15]
9	**Design Review**
10	-
11	-
12	-
13	**Client Acceptance Test**

Table 6.5: Typical schedule for course wide meetings in the multi customer capstone course (adapted from [BKA15]).

workflow as described in Figure 5.4 and then switch the roles to apply it again from the other perspective.

In the class about release management, the release coordinator shortly repeats the most important concepts of distributed version control including branch and merge management. He then introduces Bamboo as continuous integration server, shows how to configure build plans with multiple stages in Bamboo and how to deal with branches to create event based releases (see Figure 5.11). He also presents HockeyApp as delivery server, explains the differences between internal and external releases and how release notes can be configured within the workflow. He shows how crash reports and feedback reports are uploaded to the delivery server and then forwarded to the issue tracker, and explains how developers can handle feedback. In the tutorial, he interactively walks through the release management workflow in Figure 5.10 and the feedback management workflow in Figure 5.13 so that all students of the capstone course apply these workflows at least once.

6.2 Lecture

In 2015, we used Rugby's workflows to improve the collaboration between instructors and students in a university lecture "Software Engineering II: Project Organization and Management" (POM). Typically the instructor and exercise workflows are separated in such lectures, i.e. the students learn a concept in the lecture and apply it only two weeks later in an exercise, when they already forgot the concept. Rugby allows to integrate both workflows with each other in interactive classes. The students learn a concept theoretically, then immediately apply it in a small exercise and provide feedback about their progress to the exercise.

The module description of POM describes the following intended learning outcomes: Participants understand the key concepts of software project management. They are able to deal with typical problems such as writing a software project management plan, initiating and managing a software project and tailoring a software lifecycle. They are also familiar with the most important techniques of risk management, scheduling, planning, quality management, build management and release management, and apply them to solve simple problem.

In recent years, modern trends such as agile methods were only a small focus of lectures and exercise, and techniques such as distributed version control, continuous integration and continuous delivery, which software project managers need to understand to communicate with their developers, were not covered. In 2015, the teachers put a higher emphasis on agile techniques and introduced the main concepts of continuous software engineering using Rugby's workflows in lectures and central exercises with up to 450 registered students participating in the course. The lecture and exercise schedule is shown in Table 6.6.

The students learned early in the course about agile methods so that they could apply them in the exercises throughout the semester. Later, they learned about software configuration management and applied their knowledge in exercises about branch and merge management, continuous integration and continuous delivery. In 2015, the exercise concept consisted of three different types of exercises divided into individual exercises and team based exercises:

- **Individual exercises**
    1. Quizzes with multiple choice questions about previous lecture content, conducted in the beginning of the lecture as a motivation to attend classes.
    2. Interactive tutorials with detailed step by step instructions, conducted in class or as homework.

- **Team exercises**
  3. Team based exercises in a small agile project with 5 students who manage the development of a small mobile application by regularly conducting meetings, configuring tools to apply software configuration management techniques and delivering software artifacts.

The students were able to earn up to 600 exercise points for the successful participation in the exercises in order to improve their grade in the final exam of the course. This motivated the students to participate in exercises. We mapped exercise points to a specific grade improvement.

If the students earned between 120 and 239 points, they received a 0.3 grade improvement. Students with 240 to 359 points improved their grade by 0.7. 360 to 479 bonus points led to an improvement of 1.0 and more than 480 points improved the grade by 1.3. The students could earn up to 300 bonus points through team exercises, independent which students in the team exactly solved the exercise. They could get up to 82 points when successfully answering quiz questions, which repeated the lecture content, and up to 218 points through the participation in interactive tutorials. The maximum amount of individual bonus points was 300 as well.

Class	Lecture	Exercise
1	Introduction	Icebreaker
2	Project Organization	Software Project Management Plan Meeting Management
3	Software Process Models I	Agile Methods I
4	Software Process Models II	Agile Methods II
5	Kanban	Kanban
6	Usability Management	Prototyping
7	Software Configuration Management	Distributed Version Control Branch and Merge Management
8	Contracting	Review Management
9	Developing Winning Proposals	Testing and Continuous Integration
10	Capability Maturity Model	Continuous Delivery and Release Management
11	Estimation and Scheduling	Estimation and Scheduling
12	Risk Management	Demo Management and Software Theater
13	Global Project Management	Global Project Management [LKLB16]
14	Project Management Antipatterns	Repetitorium

Table 6.6: Lecture and exercise schedule of *Software Engineering II: Project Organization and Management* in 2015

6.2.1 Individual Exercises

Individual exercises built on blended, problem based, cooperative and experiential learning techniques (see Section 2.7) to increase the learning experience of the students. Students had to solve tasks on their computer over the Internet which supports the idea of blended learning. They cooperated with the instructor, tutors and neighbors to solve particular problems. They learned from their experience in these exercises and reflected about the concepts they just learned before. The list of individual exercises (except quizzes) and their corresponding exercise points is shown in Table 6.7.

Class	Exercise title	Points	Exercise description
1	Organization	5	Login to JIRA
		5	Upload image to Gravatar[8]
3 + 4	**Agile methods**	**30**	Creation of a product backlog and a sprint backlog, conduction of a sprint in JIRA
6	**Prototyping**	**30**	Creation of paper based prototypes and Balsamiq[9] prototypes
7	Distributed version control	10	Basic interaction with a git repository
7 + 8	**Branch, merge and review management**	**30**	Creation of branches, merging of branches and conduction of code reviews in Stash
9	Estimation and planning	18	Estimation with a traditional and an agile technique
10	**Testing and continuous integration**	**30**	Configuration of continuous integration and test cases in Bamboo
11	**Continuous delivery and release management**	**30**	Configuration of continuous delivery in Bamboo and in HockeyApp
13	Global project management	15	Conduction of a global project in the classroom
14	Retrospective	15	Conduction of a retrospective about lectures and exercises
Sum of exercise points		**218**	

Table 6.7: Individual exercises and the corresponding bonus points for the lecture *Software Engineering II: Project Organization and Management* in 2015

[8]Gravatar is an online service that stores images for email addresses and provides open access so that other tools can download and display this image: http://www.gravatar.com

[9]Balsamiq is an online service which allows to create digital prototypes that can be exported to a PDF and executed full screen on a mobile devices: http://www.balsamiq.com

The instructor conducted five interactive tutorials: (1) agile methods, (2) prototyping, (3) branch, merge and review management, (4) testing and continuous integration and (5) continuous delivery and release management. In these tutorials, the teacher introduced concepts and immediately applied them using small exercises. The students followed by solving the exercises on their own computer using tools such as JIRA, Balsamiq, Stash, Bamboo and HockeyApp that were available in the browser over the Internet. The students could earn 30 bonus points for the successful participation in each of these five tutorials. During the tutorial, the students either looked at the detailed slides that were handed out at the beginning of the exercise or watched how the teacher conducted the exercise on the presentation computer. In addition, tutors walked through the lecture hall, helped students and answered questions directly.

One interactive tutorial consists of three to five exercises which were decomposed into smaller tasks. In summary, the students had to solve between twelve and twenty tasks in one tutorial. The teacher synchronized the speed of the tutorial several times by asking students about their progress and by checking the number of participants and results in the tools. If more than about 90 % were able to complete a particular tasks, the teacher proceeded to the next exercise.

The student group was heterogeneous because the course was offered in multiple programs. Two distinct groups participated: (1) bachelor students in information science with a few experiences in software engineering and (2) master students in computer science with more experiences in software engineering and project management. This heterogeneity posed the challenge that students had a different velocity in completing the class exercises. To alleviate this challenge, the exercises included optional tasks specifically for more experienced students. In addition, the students had the opportunity to solve exercises as homework if they were not able to finish them in class. Some tasks and exercises were also explicitly designed as homework, e.g. when the students had to write additional test cases for the continuous integration exercise.

As an example, we describe the conduction of two exercises. In class 10 and 11, the students applied continuous integration and continuous delivery. As shown in Figure 6.4, the instructor mapped the deployment process to the service that was used, Bamboo. To simplify the exercise, each student first forked a preconfigured source code repository and cloned a preconfigured build plan. Then the students adapted and configured the build plan, fixed existing failing test cases and wrote additional test cases. A change in the requirements of the software led to a regression that was detected by Bamboo and fixed by the students so that all tests passed again at the end of the exercise and the students could deliver the software to their neighbors who played the role of test users.

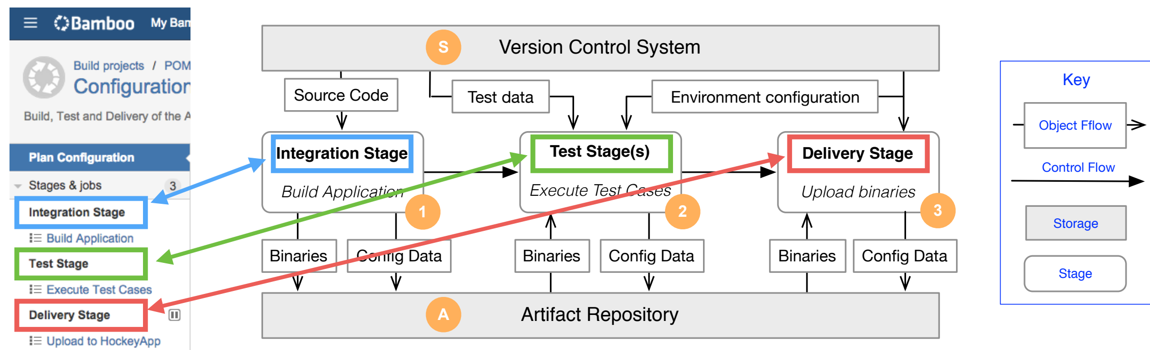


Figure 6.4: Mapping between the deployment process and stages in Bamboo

6.2.2 Team based Exercises

In addition to the individual exercises, the students participated in a small team project with five team members that accompanied the lecture. The goal of the project was that the students experience 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. In addition, they had to choose a development environment and target platform, either Android, Windows Phone or iOS.

The instructor used a script to distribute the students into teams according to their self assessment before the first exercise. The goal was to have balanced teams with respect to the skill level of the students and teams of students with matching preferences for the development environment. Without such a balancing mechanism, it would have been unfair if five experienced students build one team and another team would have had five completely unexperienced students. However, not all students participated in the team exercises. Therefore, the teams were allowed to merge teams and fill empty places in their team with other students so that the balancing was not completely enforced.

Like the individual exercises, the team based exercises also built on blended and experiential learning techniques. However, they had a stronger focus on problem based and cooperative learning. In addition, the students only received a vague description of the exercises that deliberately missed detailed instructions so that the teams have to think on their own about how to solve the exercise. This approach follows the principle “Step back and I will act” of the Chinese Proverb presented in Section 2.7.

Table 6.8 shows the schedule of the project, the list of team based exercises and their corresponding exercise points. We divided the exercises into seven phases: an initial *Sprint 0* for the project startup, five development sprints (*Sprint 1*, ..., *Sprint 5*)

each about two weeks long, and a final *Wrap Up* phase. The start and end dates of these phases were the exercises that are mentioned in Table 6.8, e.g. Sprint 0 lasted between April 17 (date of *Class 1*) and May 8 (date of *Class 4*). However, this schedule was just a rough orientation for the teams. In their SPMP, the teams could slightly deviate from this schedule, if they e.g. wanted to conduct their sprint planning and sprint review meeting on a Monday, while the exercise took place on a Friday.

Phase	Start Date	End Date	Exercise Description	Points
Sprint 0	Class 1	Class 4	Choose problem statement	5
			Choose target platform	5
			Upload icebreaker results to Confluence	10
			Write SPMP in Confluence	20
			Define product backlog in JIRA	20
Sprint 1	Class 4	Class 6	Conduct sprint planning meeting in Confluence	10
			Define sprint backlog in JIRA	10
			Design team space in Confluence	5
			Create user interface mockups	10
			Conduct sprint review meeting in Confluence	10
Sprint 2	Class 6	Class 8	Conduct sprint planning meeting in Confluence	5
			Define sprint backlog in JIRA	5
			Upload initial project source code in Stash	5
			Use branching model (continuously)	15
			Conduct sprint review meeting in Confluence	5
Sprint 3	Class 8	Class 10	Conduct sprint planning meeting in Confluence	5
			Define sprint backlog in JIRA	5
			Write an agile contract	10
			Configure continuous integration in Bamboo	10
			Write test cases and execute them on Bamboo	10
			Conduct sprint review meeting in Confluence	5
Sprint 4	Class 10	Class 12	Conduct sprint planning meeting in Confluence	5
			Define sprint backlog in JIRA	5
			Configure continuous delivery in Bamboo	10
			Release the app in HockeyApp	10
			Conduct sprint review meeting in Confluence	5
Sprint 5	Class 12	Class 14	Conduct sprint planning meeting in Confluence	5
			Define sprint backlog in JIRA	5
			Write demo scenario and script in Confluence	15
			Create software theater video with demo script	15
			Conduct sprint review meeting in Confluence	5
Wrap Up	Class 14	1 week later	Assessment of the overall project through instructor	25
			Conduct retrospective meeting in Confluence	10
			Sum of exercise points	**300**

Table 6.8: Team exercises and the corresponding exercise points for the lecture *Software Engineering II: Project Organization and Management* used in 2015

In Sprint 0, the teams chose the problem statement and the target platform, conducted a Marshmallow icebreaker [Wuj10] in the class room together with all other teams. They also wrote an initial software project management plan (SPMP) and defined the product backlog for their project in JIRA. In each development sprint, the teams conducted a sprint planning meeting, created a sprint backlog and conducted a sprint review meetings.

The focus of the lecture was on management. Therefore, it was not important for the exercises, how many backlog items the team actually realized. It was more important that the students learn how to manage the project. However, managing a software project only makes sense, if the team also develops software. To increase the learning experience of all team members, the teams had to rotate the roles for the Scrum Master, the Product Owner and the developers. Each team member then had the opportunity to play the Scrum Master role in one sprint, to play the Product Owner role in one sprint and to play the developer role in three sprints.

In addition, the teams had to conduct other exercises in the sprints as shown in Table 6.8, e.g. create user interface mockups for their application in *Sprint 1*, setting up continuous integration and writing test cases in *Sprint 3* or writing demo scenarios and demo scripts and creating a video using software theater [BKW12, XKB15] and the demo script in *Sprint 5*.

The idea was that the students learned the knowledge for these team based exercises in the lecture or in the individual exercises and that they then apply this knowledge in the team to experience typical team challenges, such as communication issues. In the final *Wrap Up*, the instructor assessed the overall project performance including the developed application and the students conducted a retrospective meeting to evaluate their project performance and to identify improvements for future projects.

6.3 Industry

In this section, we describe a case study, in which we used Rugby in industry projects in the company Capgemini[10]. We first describe the applicability of Rugby in industrial projects and then present the extended and customized versions of Rugby's release and feedback management workflows for the use in the company [KKP+15].

6.3.1 Applicability in Industrial Projects

We analyzed the development process for mobile applications in a global company with heterogeneous project environments with respect to team size and project duration. We interviewed eight project managers of Capgemini who develop software solutions for external customers in different industry sectors such as automotive, telecommunication and financial services. The answers of the interviews revealed the following *key issues* in the investigated projects before the introduction of Rugby:

There was *no standardized delivery process* for mobile applications. Projects relied on knowledge transfer from other projects. Depending on size and complexity, existing approaches might not fit. Therefore the same solutions were reinvented by multiple teams because of time constraints within mobile projects and limited communication. The *infrastructure was not sufficient* for the delivery of mobile applications. It contained the basic development tools such as an issue tracker, a version control system and an integration server, but these tools were not preconfigured and the development team was responsible for their configuration. A solution for automatic delivery was missing, some projects decided to create their own one.

Automatic testing was neglected in favor of development speed, mainly because the setup effort was too high in mobile projects, which are typically rather short living. This especially affected unit, system and integration tests. Acceptance testing with the customer was done manually, but not regularly. *Continuity was missing* in the development. Even if projects wrote tests, they rarely automated them. Continuous integration was only taken seriously by very few mobile projects. Even if projects used a build server and a delivery solution, integration and delivery were still done sporadically instead of continuously. How often a version was delivered to the customer was sometimes driven by the contract and not by development needs.

We asked the project managers why these issues occurred and why they did not yet apply a workflow for continuous integration and continuous delivery as e.g. de-

[10]Capgemini provides consulting, technology, outsourcing services and local professional services. It is presented in over 40 countries with almost 140,000 employees: http://www.capgemini.com

fined in Rugby. From their answers, we extracted the following *requirements*, which need to be addressed by project based organizations like Capgemini that consider to apply continuous delivery in their mobile projects: Mobile applications are targeting different platforms, some of them are developed natively, but in different programming languages, others use cross platform frameworks. The continuous delivery workflow should therefore *support multiple platforms*. Business critical applications require a high amount of security. Customer contracts e.g. do not allow to store applications on cloud services. *Access control* and *data privacy* are important topics and should be considered.

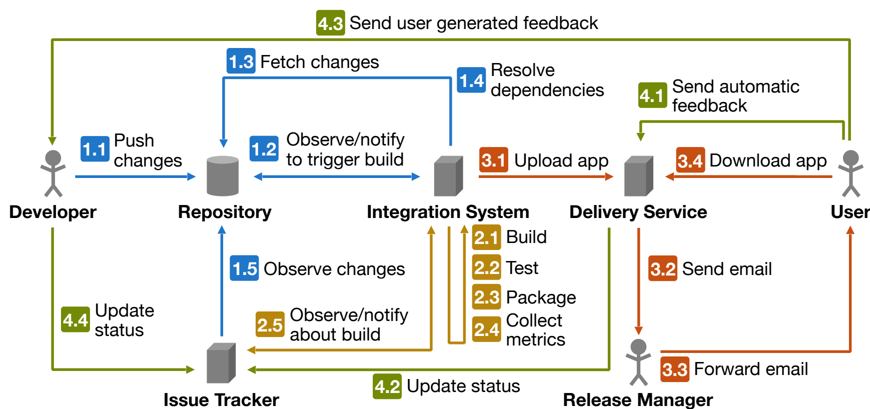
Due to pricing pressure, projects are outsourced to offshore locations. It should be possible to apply activities of the workflow in different countries for *globally distributed teams*. Applications are developed for multiple markets, thus different *legal aspects* regarding distribution need to be considered. Different project environments should be supported by *modular standardized components*, that can be easily adapted and maintained. Managers should be able to *collect metrics* about the current state of the project. Larger mobile application projects require *dependency management* for the use of external and internal frameworks. The delivery and feedback system should *distinguish* between automatically generated feedback, such as crash report and monitoring statistics, and manually created *feedback*, such as feedback reports within the application. Furthermore, it should be possible to *deliver* an application manually.

6.3.2 Extending and Customizing Rugby

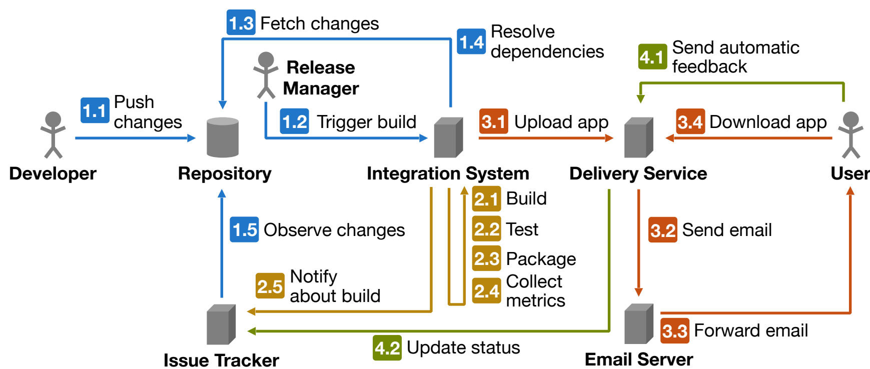
In a company with heterogeneous project environments like Capgemini, it is not possible to introduce one single, standardized workflow for all projects. Instead, project managers must be able to customize the workflow to their own needs. To address the additional requirements, we extended Rugby's release and feedback management workflow. We aimed to provide flexible activities as building blocks that can be combined depending on the individual characteristics of a given project.

Using Rugby's release and feedback management workflow as basis, we split it up into four activities: *Configuration Management* (including version control and dependency management), *Integration*, *Delivery* and *Feedback*. Each activity is optional for projects and provides customization possibilities. We also extended the activities to increase both functionality and flexibility. The resulting standard workflow and two customized workflows are shown in Figure 6.5 next to each other.

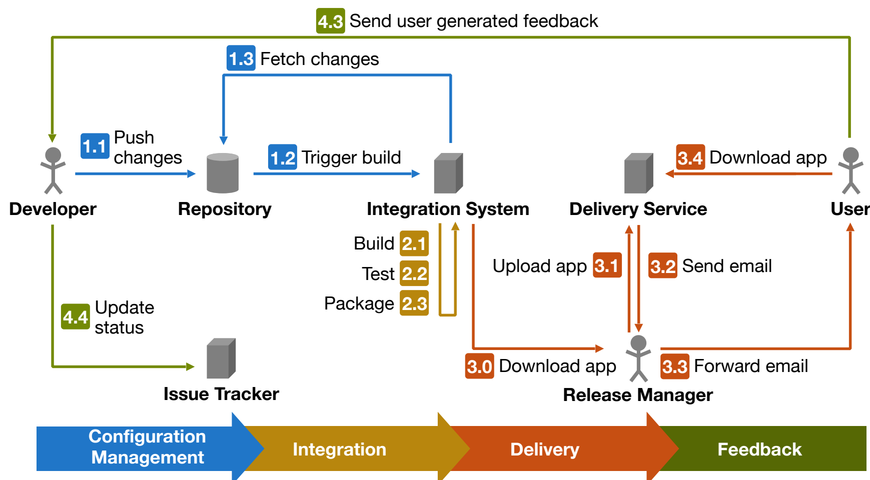
We describe scope and transitions of these four activities to provide projects with multiple variants of how each activity can be carried out (see Figure 6.5a). The ex-



a) Standard workflow, basis for customization: automatic build, automatic upload, manual distribution, both automatic and user generated feedback



b) Example of customized workflow for a large project: manual build, automatic upload and distribution, only automatic feedback



c) Example of customized workflow for a small project: no dependency resolution, no metrics, manual upload, only user generated feedback

Figure 6.5: Extended and customizable release management workflows with colors to indicate activities (adapted from [KKP+15])

tended workflow provides customization by distinguishing between mandatory and optional activities. In particular, projects can modify the workflow depending on project size, complexity, staffing, timeline and priorities. For a better understanding of how a customized version of our workflow could be instantiated, we describe two examples that we have observed at Capgemini, one for larger and more complex projects and another one for smaller and simpler projects.

Large and complex project (see Figure 6.5b): Version control uses a branching model in combination with a full fledged dependency management system. Builds are triggered manually since the integration system resolves dependencies, builds a hybrid core app first and then includes native wrappers for several platforms. It runs unit and system tests, integration tests with a backend API (Application Programming Interface) as well as automated acceptance tests for the user interface. Uploads to delivery service occur automatically and emails are automatically forwarded using lists and filters managed on an email server. Feedback is solely collected automatically from within the app usage and relayed to the issue tracker, which keeps track of changes and builds.

Small and simple project (see Figure 6.5c):

Version control uses a branching model, but dependencies are kept in the repository instead of a full fledged dependency management system. The project builds a single native app and uses an integration system that automatically fetches changes and runs a few unit and system tests. Apps are uploaded manually to the delivery service and delivery emails are forwarded manually to users. Feedback is collected from users via phone, email and personal meetings.

Additional requirements and constraints have been considered in the extended implementation of Rugby's release and feedback management workflows: The release mechanism is based on email and can be configured using mailing lists, removing the need for team members to learn new tools. A custom built, centralized delivery service avoids uploading client data to cloud services, thus addresses privacy and security concerns, and enables both manual and automated delivery.

The integration system has been rebuilt from scratch, using available infrastructure and Jenkins[11] as open source component with suitable plugins. The solution can be ordered from the central IT department of Capgemini as a turn key solution running in an isolated virtual machine. It comes with a basic configuration, but projects can adapt the entire tool set to their specific needs.

The implementation of the release and feedback management workflow requires collaboration between several individuals and departments. The organizational trans-

[11] Jenkins is an open source continuous integration server: https://jenkins-ci.org

formation necessary to introduce and subsequently fine tune this process involves detailed descriptions of aspects such as responsibilities, approval process, project-specific resources, reporting structures etc. However, this is not part of our discussion and covered by others such as Humble [HF10] and Poppendiek [PP06].

Chapter 7

Evaluation

> "I think it's very important to have a feedback loop, where you're constantly thinking about what you've done and how you could be doing it better."
>
> — Elon Musk

In this chapter, we describe the evaluation of Rugby in university and industry. First, we state nine hypotheses in Section 7.1, two main hypotheses, five derived hypotheses for software engineering and two derived hypotheses for education. Then, we present the study design of six qualitative and three quantitative evaluations in Section 7.2. We present and interpret the findings in the evaluations for the seven derived hypotheses in Section 7.3. Finally, we describe the limitations of the nine evaluations in Section 7.4 and summarize the findings with respect to the hypotheses in Section 7.5.

7.1 Hypotheses

Rugby can be used in multiple contexts, in particular in continuous software engineering and continuous education. Our main hypotheses are:

H1 Rugby allows to reduce the delay between development and usage in software projects.

H2 Rugby allows to reduce the delay between lectures and exercises in education.

Rugby's process model provides an event model that increases the frequency of delivery and feedback, which reduces the size of the content – software changes in development – and increases the amount of interaction. Through the increased interaction with a smaller content size, the collaboration between developers and users and between instructors and students is improved. We derive the following five hypotheses for Rugby in software development projects:

H 1.1 Review: Developers increase their code quality with Rugby's review management workflow.
H 1.2 Release: Rugby's release management workflow reduces the effort required to create a release.
H 1.3 Feedback: Rugby's feedback management workflow increases the quality of feedback.
H 1.4 Frequency: The use of Rugby's process model increases the number of reviews, releases and feedback reports.
H 1.5 Understanding: The presentation of event based releases as executable prototypes improves communication and understanding between project participants.

We hypothesize that continuous peer reviews in Rugby's review management workflow allow developers to improve the code quality and increase the understanding about code and design within the team (H1.1). This ensures that only high quality is present in released product increments. We hypothesize that the automation of testing, integration and delivery in Rugby's release management workflow reduces the effort of these activities and improves the efficiency and the productivity of the team (H1.2). Rugby's feedback management workflows enables automated feedback such as crash reports with context information and the built in feedback approach motivates users to provide feedback. Therefore, we hypothesize that the quality of feedback is improved (H1.3).

Each of Rugby's three workflows increases the number of produced outcomes, therefore we hypothesize that the number of reviews, release and feedback reports increases when Rugby is used (H1.4). Rugby's release management workflow allows the creation of event based releases within an iteration. We hypothesize that event based releases used as executable prototypes improve the communication and the understanding between project participants (H1.5).

Rugby's process model is extensible to education where its event model reduces the size of the content – theoretical concepts in education – and increases the amount of interaction between educators and learners. We derive the following two hypotheses for Rugby in eduction:

H 2.1 Learnability: Rugby's workflows can be effectively taught by an instructor in university capstone courses and lectures within one semester.
H 2.2 Scalability: Rugby enables instructors to scale exercises in large class rooms with more than 100 students.

We hypothesize that Rugby's workflows for review, release and feedback management can be taught to students in the university environment within one semester, so

that they feel comfortable with and want to use the workflows in future projects (H2.1), even if they did not use these workflows before. We further hypothesize that Rugby enables the conduction of exercises with more than 100 students in the class room while reducing the effort of the exercises (H2.2), because it automizes exercises and provides smaller chunks of theory intermixed with exercises.

7.2 Study Design

We validate the seven derived hypotheses with qualitative and quantitative evaluations. Table 7.1 shows an overview of six formative and qualitative evaluations (E1, ..., E6) and their mapping to the hypotheses. It provides information about the evaluation method, the corresponding case study (C1, C2, or C3) as described in Chapter 6, in which we conducted the evaluation, the covered topics of the study, the pool of participants and the response rates.

ID	Year(s)	Evaluation method	Case Study	Topics	Participants Pool	# Responses (Rate)	Hyp
E1	2013	Questionnaire	C1	Release management Feedback management	90 students[1]	41 (46 %)	H1.2 H1.3
E2	2014	Questionnaire	C1	Review management	90 students[2]	81 (90 %)	H1.1
E3	2011 - 2014	Questionnaire	C1	Improvement of technical and soft skills	301 students[3]	178 (59 %)	H2.1
E4	2014	Questionnaire	C3	Release management Feedback management	8 project managers	8 (100 %)	H1.2 H1.3 H1.4
E5	2015	Personal interviews	C1	Review management	90 students[4]	18 (20 %)	H1.1
E6	2015	Questionnaire	C2	Review management Release management Feedback management	423 students	223 (53 %)	H2.1 H2.2

Table 7.1: Overview of the formative and qualitative evaluations between 2013 and 2015

We conducted our first evaluation (E1) about release management and feedback management in 2013. We wanted to understand whether release management re-

[1]The pool of 90 students includes developers and coaches in the SS 2013 course, compare Table 6.2.
[2]The pool of 90 students includes developers and coaches in the SS 2014 course, compare Table 6.2.
[3]The pool of 301 students includes developers and coaches in the courses SS 2011, SS 2012, SS 2013 and SS 2014, compare Table 6.2. Some students participated twice, first as developer and then in a later iteration as coach.
[4]The pool of 90 students includes developers in the WS 2014/15 course, compare Table 6.2.

duced the effort to create a release (H1.2) and whether feedback management increases the quality of feedback (H1.3). In 2014, we evaluated review management (E2) in the capstone course to understand its impact and whether students can conduct peer code reviews in order to improve their code quality (H1.1).

In 2014, we conducted a quasi experiment (E3) about the improvement of technical skills and soft skills of students in the four capstone courses between 2011 and 2014 to investigate whether Rugby's workflows can be effectively taught (H2.1). We evaluated (E4) release management and feedback management in industry in 2014 to validate the generalizability of our findings in the university with respect to H1.2, H1.3 and H1.4.

In personal interviews (E5) about review management in 2015, we addressed the limitations of our earlier evaluation on review management (E2), that we did not ask the participants on the review purpose. The personal interviews aimed to find additional insights about purpose and quality improvements through reviews. The questionnaire (E6) in the lecture allowed us to investigate the subjective learning experience of the students (H2.1) and the scalability of Rugby's workflows in the class room (H2.2).

Table 7.2 shows an overview of three formative and quantitative evaluations (E7, E8, and E9) and their mapping to the hypotheses.

ID	Year(s)	Case Study	Topics	Quantity	Hyp
E7	2012 - 2014	C1	Release management Feedback management	32 projects	H1.4
E8	2014 - 2015	C1	Review management	33 projects	H1.4
E9	2015	C2	Grade improvement through exercise participation	294 students	H2.1, H.2.2

Table 7.2: Overview of the formative and quantitative evaluations between 2012 and 2015

We conducted E7 to measure the frequency of delivery and user feedback (H1.4) in order to understand whether the introduction of release management and feedback management workflows increased the frequency. With E8, we wanted to find out whether the frequency of code reviews increased (H1.4) through the introduction of the review management workflow. The evaluation E9 helped us to validate the results of the qualitative evaluation (E6) of the subjective learning experience in the lecture, because we could measure the correlation between exercise participation and the exam grade, which is objective data that is not influenced by the personal opinion of the students. In the following, we describe the study design of all evaluations (E1, ... E9) in more detail.

E1: Qualitative Study in Capstone Course in 2013

The questionnaire (E1) in the SS 2013 capstone course investigated the students' learning and the benefits of the release management and feedback management workflows [KA14]. After the course, we invited 90 students, developer and coaches including the release managers of each team, to participate in an online questionnaire. The anonymous questionnaire consisted of 23 questions, took about 15 minutes and was not mandatory for the students.

15 questions of the survey were closed questions with predefined answer possibilities. Eight questions were open questions with a large answer field where the respondents could write their opinion. We conducted the survey in August 2013 during the exam phase and gave the students four weeks to complete it. After two weeks we sent out a reminder email to all students. We received 41 valid responses out of 90 students, about 30 % of the answers were given by undergraduate and about 70 % by graduate students. At least three students of each team participated in the questionnaire.

The questionnaire covered questions with respect to release management and feedback management, in particular about continuous integration and continuous delivery. We categorized the questions into five sections: personal data (4 questions), knowledge acquisition (7 questions), workflow usage (5 questions), workflow benefits (3 questions) and personal experience (4 questions). In the personal data category, we asked about the students about their major, their semester, their team and their degree. The knowledge acquisition category contained questions about prior experience with continuous delivery and evaluated whether they understood the main concepts.

In the workflow usage category, we evaluated our teaching methods, i.e. which teaching approaches the students used the most to understand the workflows and how much knowledge they gained through each of them. The workflow benefits category asked how often participants used the activities for version control, continuous integration, continuous delivery and obtaining user feedback. In the personal experience category, we asked if they understood the main goals and benefits of continuous delivery and user feedback.

E2: Qualitative Study in Capstone Course in 2014

The questionnaire (E2) in the SS 2014 capstone course investigated the benefits of the review management workflow and whether students improved their coding skills [KBB16]. After the course, we invited 90 students, developer and coaches including the review managers of each team, to participate in an online questionnaire. The

anonymous questionnaire consisted of 29 questions, took about 20 minutes and was not mandatory for the students.

21 questions of the survey were closed questions, eight questions were open. Five open questions and one closed question were conditional questions, i.e. they were only shown to participants that chose specific answers in a previous question. One example is that we only asked students why a merge conflict happened, if they answered *Often*, *Sometimes* or *Once* in the previous question how often a merge conflict happened. If they answered *Never*, then the question was not shown.

We conducted the survey in August 2014 during the exam phase and gave the students four weeks to complete it. We created personalized tokens and send them to each participant to increase the response rate. The survey tool LimeSurvey[5] guarantees that the answers are anonymous by strictly separating the token and the answers tables in the database. After one week we sent out a reminder email only to the students who did not yet participate. After another week, we sent a second reminder. With these mechanisms, we could receive 81 valid responses out of 90 students, about 30 % of the answers were given by undergraduate and about 70 % by graduate students. At least four students of each team participated in the questionnaire.

The questionnaire covered questions with respect to review management. We categorized the questions into eight sections: personal data (3 questions), knowledge (3 questions), coding support (3 questions), branching model (3 questions), merge requests (3 questions), review workflow (6 questions), review workflow problems (4 questions), personal experience (4 questions). In the personal data category, we asked the students about their major, degree and team. In the knowledge category, we asked about existing knowledge before the course and how much knowledge participants gained through the course. The coding support category included questions about the use of coding guidelines in the course. In the branching model category, we asked about the benefits of branch and merge management and how often participants experienced mistakes such as accidentally deleting a branch and why this happened.

The merge request category included questions about the usage frequency and the benefits of merge requests. In the review workflow category, we asked whether students agree to benefits, such as improved quality, and whether and why participants bypassed the workflow. The review workflow problems category asked whether students experienced specific problems, such as simple or complex merge conflicts and, if this was the case, how they could solve the problem, why the problem occurred and how it could be prevented. In the personal experience category, we asked whether

[5]http://www.limesurvey.org

students would use the workflow again, what worked well in their team and what was the main issue in their team with respect to review management.

E3: Qualitative Study in Capstone Courses between 2011 and 2014

The qualitative quasi experiment (E3) between 2011 to 2014 investigated the students' improvement of technical and soft skills [BKA15]. We performed a quasi experiment to analyze the introduction of release management in 2013 and the introduction of code review workflows in 2014 as interventions. We used a five point Likert type scale with the answers *strongly disagree*, *disagree*, *neutral*, *agree*, *strongly agree* to measure either negative, neutral or positive responses. The anonymous questionnaire consisted of four questions, took about 5 minutes and was not mandatory for the students. Two questions were closed, two questions were open.

We conducted the survey in November 2014 and gave the students three weeks to complete it. We created personalized tokens and send them to all participants of the four capstone courses to increase the response rate. The survey tool LimeSurvey guarantees that the answers are still anonymous by strictly separating the token and the answers tables in the database. We sent two reminders to the students who did not participate until this point in time. We invited 301 students to participate in an online questionnaire and received 178 responses. The overall response rate was 59 % (2011: 33 %, 2012: 56 %, 2013: 57 %, 2014: 71 %). While the were more answers from later years, we received enough results from earlier years to be able to compare the results. We combined the responses of the results into a three point Likert type scale with positive responses (*strongly agree* and *agree*), neutral and negative ones (*strongly disagree* and *disagree*) to minimize positive and negative outliers.

The questionnaire asked in which year the student participated in the course. The second question asked the student to rate his personal technical skill improvements in eight software engineering categories, such as requirements elicitation, programming and release management. It also asked the student to rate his personal soft skill improvement in four categories such as communication and team work. The other two questions were open and asked about what the student liked in particular in the course and what could be improved.

E4: Qualitative Study in Industry in 2014

The questionnaire (E4) of eight industry projects in 2014 investigated the use of release management and feedback management at the company Capgemini [KKP$^+$15]. We designed the questionnaire as an expert interview in order to benefit from the

technical and domain expertise of participants. We chose project managers as questionnaire participants according to their technical, process and explanatory knowledge, who have multiple years of experience in the mobile domain and insight into the workings of mobile projects. All project managers have an understanding of and preliminary experience with agile methodologies and the standard components of our integration system are already known within the company. However, there is little experience with automated delivery in general and no experience with our custom built delivery service.

After the introduction of Rugby in these projects, we asked the eight project managers to participate in a survey which consisted of 32 questions and took about 30 minutes. 26 questions of the survey were closed, six were open. All eight project managers participated.

The questionnaire covered questions with respect to release management and feedback management, in particular about metrics, integration, delivery and feedback. We categorized the questions into five sections: intro (3 questions), project (11 questions), integration (9 questions), delivery (6 questions) and feedback (3 questions). We asked participants how fast, well or easy the process of build, test, delivery and feedback collection were before and after the introduction of our solution. We also estimated the project complexity using metrics and feedback. Additionally, we included informal questions to further elaborate on the setup, processes, and situation in the project and to evaluate the project managers' opinion about our solution.

E5: Qualitative Study in Capstone Course in 2015

The personal interviews (E5) in the capstone course in WS 2014/15 investigated the environment, the technique, properties and results of reviews as well as the tool quality in Rugby's review management workflow. After the course, we invited review managers and developers to participate in a personal interview which took between 60 and 90 minutes for reviewers and about 30 minutes for developer. The time depended on the discussion and on how many details the interviewees explained.

We used a structured interview manual and separate questionnaires for the two types of roles. Interviewing the two groups is important for understanding both sides of the review process. Whereas the reviewer questionnaire focuses on the review influence factors and its execution, the developer questionnaire focuses more on the perceived review byproducts such as potential code quality improvement, development process disturbance and knowledge gain.

The participation was voluntary and because of its length, it was limited to a subset of the students. We conducted each interview face to face with eleven reviewers and

five developers. The reviewer questionnaire covered 13 categories with multiple questions per category. The developer questionnaire included a subset of the questions of the reviewer questionnaire in eleven categories.

E6: Qualitative Study in Lecture in 2015

The questionnaire of the lecture in 2015 (E6) investigated the students' improvements and confidence in the techniques that we applied in the individual and team based exercises. After the lecture, we invited 423 students, who were registered for the lecture, to participate in an online questionnaire. The anonymous questionnaire consisted of six closed questions, took about five minutes and was not mandatory for the students.

The first two questions were about personal data, the field of study and degree. The third question asked whether the students participated in six individual exercises, while the fourth question asked whether the students applied the same six techniques in their team project. The last two questions used a five point Likert type scale with the answers *strongly disagree*, *disagree*, *neutral*, *agree*, *strongly agree* to measure either negative, neutral or positive responses. The fifth question measured if students were able to improve their skills in these six techniques and the sixth questions measured if students are confident to apply these six techniques in their next team project.

We conducted the survey in July 2015 and gave the students two weeks to complete it. We created personalized tokens and send them to all participants of the four capstone courses to increase the response rate. The survey tool LimeSurvey guarantees that the answers are still anonymous by strictly separating the token and the answers tables in the database. We sent two reminders to the students who did not participate until this point in time. We received 223 responses (53 %). We combined the responses of the results into a three point Likert type scale with positive responses (*strongly agree* and *agree*), neutral and negative ones (*strongly disagree* and *disagree*) to minimize positive and negative outliers.

E7: Quantitative Measurement in Capstone Courses between 2012 and 2014

The quantitative measurement (E7) in the three capstone courses SS 2012, SS 2013 and SS 2014 focused on numbers of the release management and feedback management workflows that we introduced after two thirds of the capstone course in 2012 and from the beginning in 2013. We used the git command line to count commits and branches and used the REST API of Atlassian Bamboo to count the number of builds and releases. We had to manually count the number of downloads and feedback reports, because no REST API was available. We then took the average of these

numbers per team. We measured the values after the courses was finished. Some values in 2012 are not available, therefore we estimated an generous upper number.

E8: Quantitative Measurement in Capstone Courses between 2014 and 2015

The quantitative measurement (E8) in the three capstone courses SS 2014, WS 2014/15 and SS 2015 focused on numbers of the release management workflow that we introduced in 2014 from the beginning. We used the REST API of Atlassian Stash in a custom command line application to count for each merge requests the number of related commits (excluding merge commits), the number of comments, the number of reviewers and the interval, i.e. the time between the merge request was opened and finally merged. We then took the average of these numbers per merge request and also calculated the average number of merge request per team. We measured the values after the courses was finished.

E9: Quantitative Measurement in Lecture in 2015

The quantitative measurement (E9) in the lecture in 2015 focused on the correlation between exercise participation and the final grade of the student. In order to upload the final grades into the lecture and exam management tool TUMOnline, we inserted all exam results and the exercise participation into an Excel sheet, where all students were listed that participated in the final exam. The Excel sheet therefore included the final exam grade and the number of exercise points (divided into quizzes, individual exercises and team based exercises) for each student. We used the Excel sheet to group the students after the exercise points into four categories and calculated the grade point average for each category.

7.3 Findings

In this section we describe the findings according to the seven derived hypotheses. We also show, in which evaluation we found the findings, interpret the results and indicate whether the findings support the hypothesis or not.

7.3.1 Review

Hypothesis H1.1 states that developers increase their code quality when using Rugby's review management workflow. To evaluate this hypothesis, we interviewed developers in the capstone course (E5). Figure 7.1 shows interview findings from capstone course participants who ranked their motivation for conducting reviews and the achieved benefits they perceived. Ensuring architecture compliance and improving code readability and maintainability were the main motivators, and likewise, the most prominent benefits of branch based reviews. The interviewees agreed that reviews had a bigger impact with sharing system knowledge among the team than initially believed. Educating novice developers about best practices and quality standards was a strong motivator and benefit. While reviewers did find defects using reviews, it was not more than they had expected.

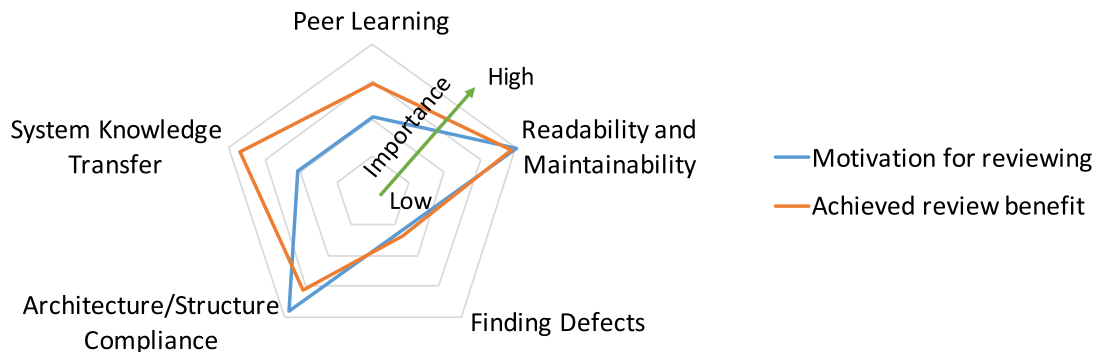


Figure 7.1: Answers in personal interviews (E5): The developers' view on motivations and benefits of code reviews

Rugby's review management and release management workflows both include the use of a branching model. Figure 7.2 shows that the participants see the benefits in using a distributed version control system combined with a simple branching model. 98 % of the team members think that the branching model helped to work with multiple persons on the same codebase. More than 40 % of the developers used features branches to develop multiple prototypes for one functionality. We asked whether they experienced the benefits of continuous integration (E2).

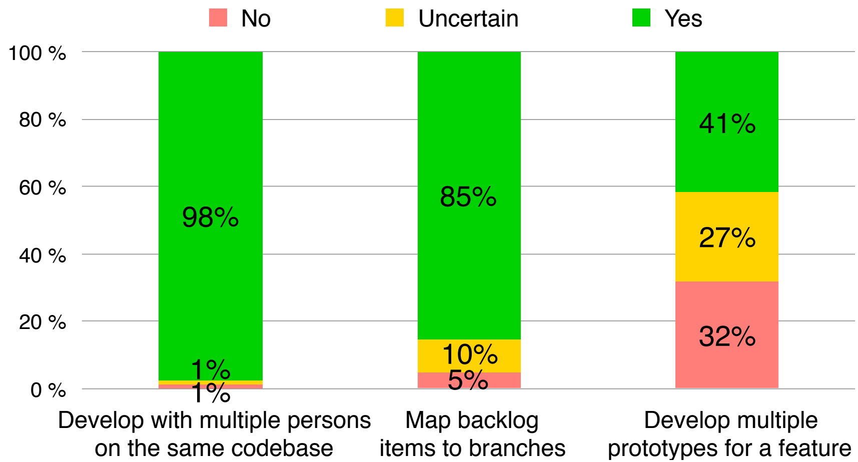


Figure 7.2: Answer of questionnaire participants (E1): Rugby's branching model helps to

To evaluate possible workflow challenges, we wanted to know whether students encountered specific problems while using the workflow, e.g. if they encountered simple or complex merge conflicts (E2). Figure 7.3 shows that simple merge conflicts occurred often. Students reported that they could easily resolve them.

Complex merge conflicts such as when multiple developers worked on non mergeable files, happened less often, nonetheless, three out of four students experienced them. Some students reported that they had problems resolving complex merge conflicts on their own. However, they were also able to solve this challenge by asking experienced team members for help. We also asked how to prevent such errors and some common answers were to improve team communication, to only change non mergeable files on the development branch and to pull changes from development into feature branches more often. One important answer was that they should try to

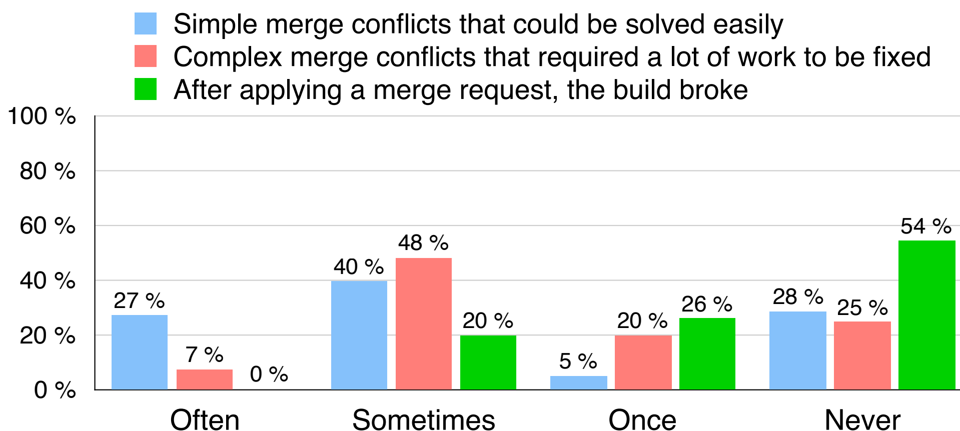


Figure 7.3: Answer of questionnaire participants (E2): How often did you encounter the following branch and merge management problem?

minimize the lifetime of branches. We conclude from these findings that students can handle challenges that Rugby's review management workflow poses.

A problem of the code review workflow is that the additional step of the quality gate might slow down the progress of the team. In situations with high time pressure such as right before a sprint review meeting, developers might not thoroughly review a merge request to avoid the additional effort. However, only 22 % of the respondents believe the workflow affected the team's progress as shown in Figure 7.4.

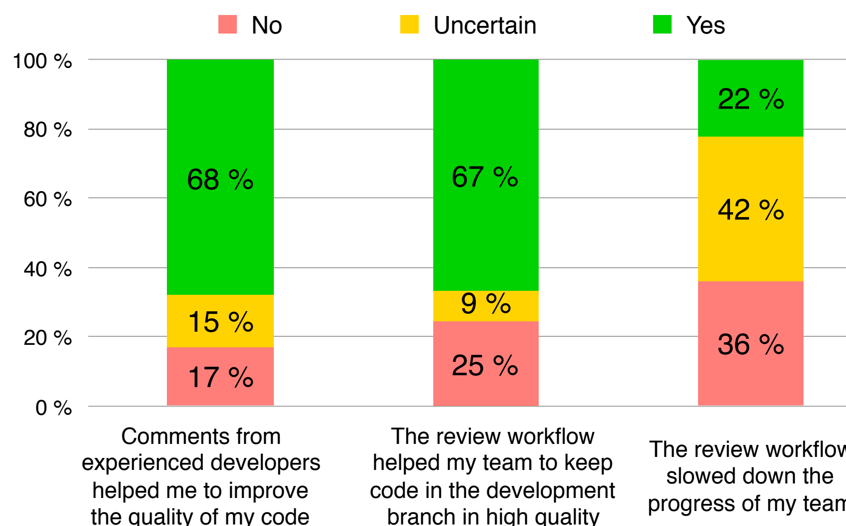


Figure 7.4: Answer of questionnaire participants (E2) about review management

Another focus in our evaluation was the learning experience through the code review workflow. Figure 7.4 shows that more than two thirds of the developers could improve the code quality of their own code with the feedback of experienced developers, because they used Rugby's review management workflow. They reported about mentoring relationships between experienced and inexperienced developers where both, developer and reviewer improved their skills. We received the same results when we asked whether the code review workflow helped the team to sustain good code quality in the development branch. As shown in Figure 7.4, two thirds think that the workflow had an important impact in preventing the negative effects of the broken window theory [HT00].

We asked whether developers would use the review workflow again in future projects. Only one student replied that he would not do so (E2). Figure 7.5 shows that 43 % of the developers definitely want to use the workflow again, 27 % very likely and 19 % likely. We found anecdotal evidence that Rugby's review management workflow leads to better code quality. We conclude that our findings support hypothesis H1.1. More-

over, developers learned about different parts of the system from reviewing their peers, a finding other researcher also found in code review studies in industry [RB13].

7.3.2 Release

Hypothesis H1.2 states that Rugby's release management workflow reduces the effort required to create a release. To evaluate H1.2, we first asked the participants if they would use Rugby's workflows in future projects. 83 % want to definitely or very likely use the branching model, 63 % will very likely use continuous integration and continuous delivery again (E1 and E2).

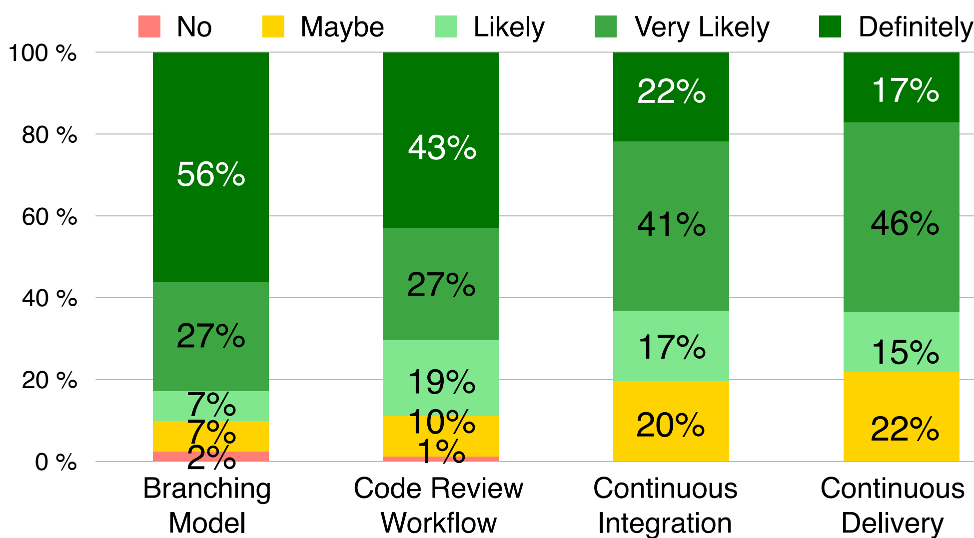


Figure 7.5: Answer of questionnaire participants (E1 and E2): Would you use this Rugby workflow in future projects?

In the industry evaluation (E4), we found the following results: number of steps required for delivery (e.g. building, signing, packaging, uploading the app and then notifying users) reduced from 5 to 1 (median). Projects with a more complicated release process were even able to reduce complexity from 10 to 1 (median). Likewise, involvement of team members in the delivery process was reduced by 25 % (median), in most cases requiring only one release manager.

Before the introduction of an automated process, deliveries required one hour up to an entire business day. Projects were able to reduce this duration to 5 minutes (median). These time savings implicitly improve cycle time and free up project members to address other tasks [HF10]. As a side effect, projects reportedly plan to involve more external users in the future, now that complexity of delivery no longer increases with number of users. We found anecdotal evidence that Rugby's release management

workflow reduces the time required to create releases. Moreover, developers appreciate the workflows and like to use it again in future projects. We conclude that these findings support hypothesis H1.2.

7.3.3 Feedback

Hypothesis H1.3 states that Rugby's feedback management workflow increases the quality of feedback. To evaluate H1.3 in the industry case study (E4), we wanted to know whether applying Rugby's release and feedback management does not only save time and effort but also increases the quality of feedback that can be collected from users during the development process. Because actual quantity of feedback is hard to measure and compare, we counted the frequency of collection. Our results show which channels are utilized for feedback collection: For weekly feedback collection, 90 % of projects use email and phone, followed by meetings (50 %) and software tools (25 %).

40 % of projects supplement this with additional meetings, virtual meetings or custom tools on a less frequent basis. Figure 7.6 shows that industrial projects prefer channels that yield unstructured feedback and state ease of use as the primary reason for this. Reportedly, channels that facilitate personal communication also yield the best feedback quality. More frequent delivery as well as better integration between tools also yield more defined feedback. These findings correspond to the ones in university capstone courses that we reported in [KA14].

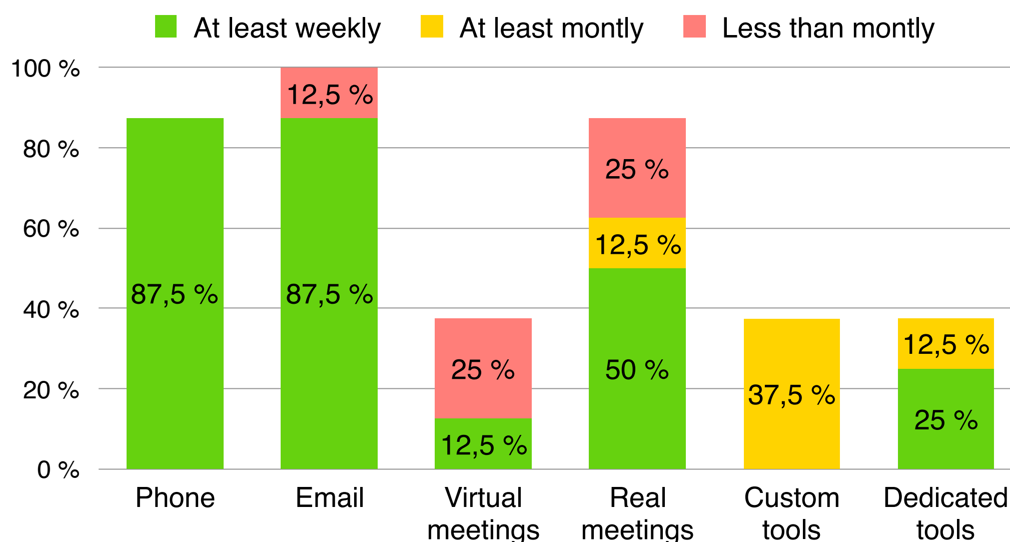


Figure 7.6: Answers in questionnaire (E4): Utilization of feedback channels and frequency of feedback collection

Figure 7.7 shows the perceived quality of feedback across the used channels. In the comparison before and after the introduction of Rugby's feedback management workflow, we found the same results for the quality of **manual** feedback. However, the quality of **automated** feedback, such as crash reports, is increased by Rugby's feedback management workflow, because additional context information such as the stack trace or the previous interaction steps are attached.

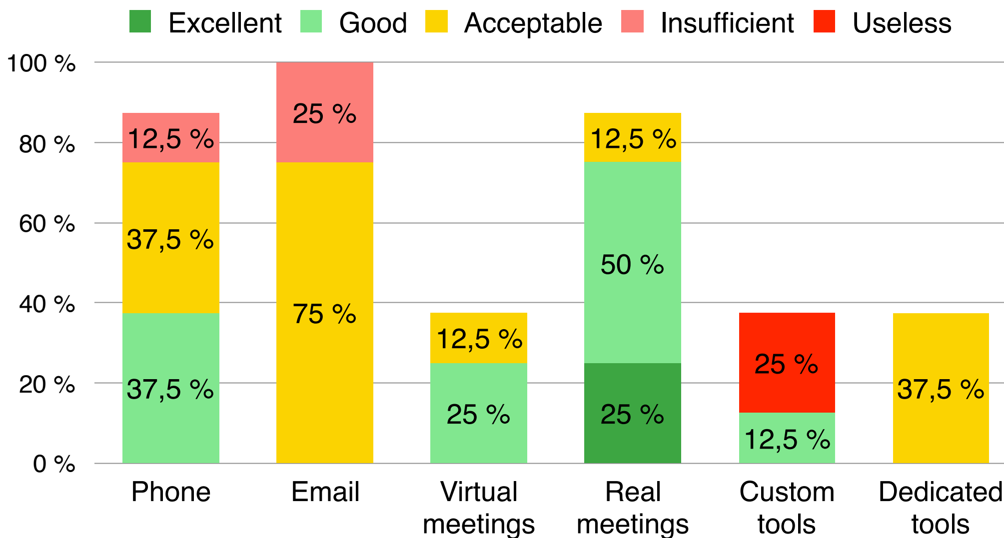


Figure 7.7: Answers in questionnaire (E4): Perceived quality of feedback across feedback channel (where used)

However, the studies do not yet show that the quality of manual feedback reports has improved. We think that increasing the quality of manual feedback is an organizational issue. Our evaluation was shortly after the introduction of Rugby in these projects and it takes more time until the project participants in industry get used to the feedback mechanism and until they actively pull user feedback with integrated feedback mechanisms, i.e. *Dedicated Tools* in Figure 7.7. In addition, it is easier to communicate feedback synchronously in meetings instead of typing it into the feedback form in the application, because direct communication is faster and less error prone. Developers can ask clarification questions directly and the user can explain the problem by showing it in the application.

We found anecdotal evidence that Rugby's feedback management workflow increases the quality of **automated** feedback through usage context. However, we did not find evidence, that the quality of **manual** feedback increases. We conclude from these findings, that hypothesis H1.3 is supported partially.

7.3.4 Frequency

Hypothesis H1.4 states that the use of Rugby's process model increases the number of reviews, releases and feedback reports. To evaluate H1.4, we quantitatively measure these numbers and also asked about them quantitatively in a questionnaire (E1). Figure 7.8 shows that 59 % of the developers think that Rugby's Release Management workflow leads to more releases compared to a manual delivery process. 37 % think that they could obtain more feedback and 44 % think they obtain better feedback from their users by applying continuous delivery.

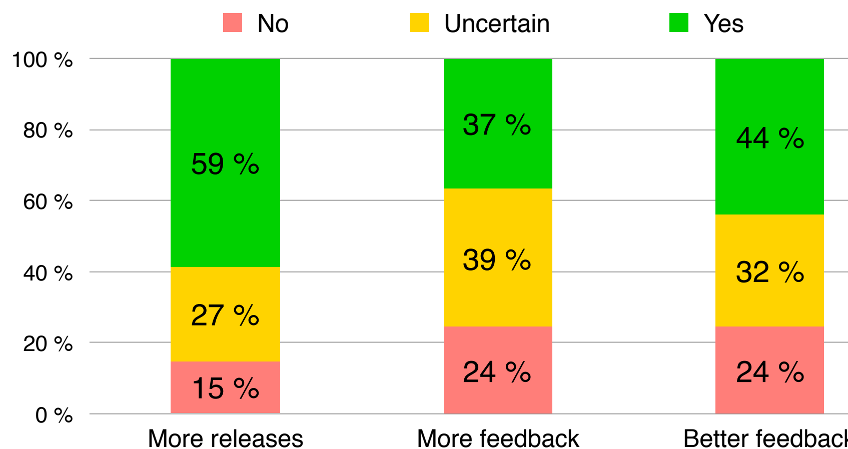


Figure 7.8: Answer of questionnaire participants (E1): Rugby's release management workflow leads to

We measured the use of branches, commits, builds, releases, downloads and feedback reports on average per team in the capstone courses between 2012 and 2015 and estimated these numbers for the capstone course in 2011. The tools that we use for Rugby's reference implementation allowed us to count these numbers by accessing a REST (Representational State Transfer) API. As shown in Table 7.3, the number of branches significantly increased in 2013 from 2 to 26 on average per team when we first introduced the branching model, while the number of commits in the version control system only slightly increased by 15 %. In 2013, more than 75 % of all commits led to a build in the continuous integration server while in 2012 only 15 % of all commits led to a build. This is caused by the fact that the build server was available from the first day in 2013 whereas in 2012 it was only available after two third of the project course.

More than 94 % of the builds were successful in 2013, because the coaches cared more about always having an executable prototype which can be presented in a meeting. Consequently, the absolute number of executable prototypes delivered to the customer is three times higher in 2013 because the teams were able to deliver releases from the first day. In 2012, we did not measure the number of downloads, crash re-

ports and feedback reports. However, in 2013 each delivered build was downloaded 2.5 times on average. Crash reports and the built in feedback were used to create 14 reports on average per team.

Year	Version Control		Release Management			Feedback Management	
	Branches	Commits[6]	Builds created	Builds green	Releases	Downloads	Feedback reports
2011	1	<200 *	0	0	<2 *	<10 *	<5 *
2012	2	500	76	56	15	<10 *	<5 *
2013	26	575	440	414	49	126	14
2014	55	728	637	591	64	136	27

Table 7.3: Measurements of average use of version control, release management and feedback management workflows per team in capstone courses (E7)] between 2011 and 2014 (adapted from [BKA15]). Numbers with a star * are estimated.

In 2014, the number of branches further increased because we introduced and used a code review workflow based on branch and merge management. The number of commits also increased because we asked the developers to produce smaller commits for the code reviews. This also led to more builds released into the delivery environment. While the number of downloads only slightly increased, the number of feedback reports using the built in feedback mechanism increased by 96 % in 2014. This means that the teams received 27 feedback reports on average in 2014. Around half of these reports were crash reports pointing to bugs in the source code, the other half of these reports were feature and design requests, mostly given by non technical customers or users of the developed applications.

We observed that more users recognized the benefits of providing feedback directly while trying out the application. About two third of the releases were event based within the sprint and about two third of the feedback reports originated from these event based releases. Customers preferred to give feedback to product increments produced at the end of the sprint directly to the team in the sprint review meetings. Developers reported that event based releases required only minor effort and provided a great opportunity to confirm the realization of requirements or to obtain early feedback so that they could save time and effort.

Table 7.4 shows the number of repositories, merge requests, comments, commits for three capstone courses, which had eleven projects each. Additionally, we computed the interval (i.e. the amount of time in hours) for each pull request, from when it was created until it was merged. We also included the average measures per pull request

[6]We counted all commits including merge commits.

as well as standard deviation and coefficient of variation to compare the results. We filtered out declined reviews and did not count merge commits or commits which were not referenced in merge requests (e.g. because someone directly committed to the development branch or to the master branch).

In the course in SS 2014, the teams created and approved 1053 merge requests, on average 96 pull requests per team. In the course in WS 14/15, the average per team was 74, while in SS 2015 it was 97. This indicates the teams' frequent use of the review workflow. There were differences between teams, which we attribute to the following three reasons: (1) The number of pull requests depends on the partition of requirements into sprint backlog items such as features; some teams came up with a high amount of small features, while others created a small amount of large features. (2) Some teams also used hotfix branches for each bug, which then consisted of only a very small amount of changes. (3) A few teams had a larger codebase than others.

Course	# Repos	# Merge Requests	Comments				Commits[7]				Interval		
			#	avg	sd	cv	#	avg	sd	cv	avg	sd	cv
SS 2014	3.2	96	143	1,49	± 4,03	2,70	400	4,18	± 6,05	1,45	11,19	± 22,49	2,01
WS 14/15	2.9	74	136	1,83	± 3,93	2,15	316	4,24	± 5,37	1,27	16,62	± 27,87	1,68
SS 2015	3.0	97	145	1,50	± 5,03	3,35	373	3,85	± 5,37	1,40	12,88	± 25,36	1,97

Table 7.4: Measurements of average number or reviews, comments and commits per team in capstone courses (E8). avg = average per merge request, sd = standard deviation, cv = coefficient of variation.

Some teams made little use of comments in pull requests because they worked together in the same room or used other channels to communicate the feedback. Other teams used comments quite often when they worked distributed. Most teams had a similar number of commits per pull request (between three and five). Only a few teams had a lot more commits per pull request since they used larger features that needed more code changes. Then, multiple features branches were open for a long time because the developers needed more time to actually implement and review the functionality of the feature. This in turn leads to longer review intervals and a higher chance for merge conflicts, although most teams managed to review and approve pull requests within less than one day on average. Only a few teams needed more time due to fewer reviewers or larger amounts of changes.

[7] We only counted commits referenced in the merge request, which were not merge commits. Commits directly to the development branch or directly to the master branch were not counted. Therefore the number is smaller than in Table 7.3.

We found anecdotal evidence that Rugby increases the frequency of reviews, releases and feedback reports and therefore conclude that these findings support hypothesis H1.4.

7.3.5 Understanding

Hypothesis H1.5 states that Rugby's increase of presentations of event based releases as executable prototypes improves communication and understanding between project participants. To evaluate H1.5, we investigated the use of executable prototypes in the capstone courses. In capstone courses before 2012, where the teams did not use executable prototypes as communication mechanism, the synchronization between developers in the project teams took a lot of time. In addition, the synchronization of multiple projects on project management level also took too much time, because the project leaders and the coaches were not familiar with the other projects. Therefore, they were not able to report the project status in a short amount of time in the weekly meetings.

This was also the feedback we received from some of the project leaders and coaches in 2012. Often the participants did not focus on crucial points and discussed issues too long which were not important for all meeting participants. This communication problem was intensified because of the multiplicity of problem statements and the different technical challenges in each of the projects in the project based organization of the capstone course. If a project leader talks about the status of his team without the ability to visualize it, the others can hardly follow and understand it.

After we introduced executable prototypes as main communication mechanism in meetings in 2013, project leaders and coaches were able to communicate their team status, i.e. what their team achieved since the last meeting, in significant less time. They downloaded the last event based release as executable prototype to one demo device and presented it on the shared meeting screen to all others. By showing the differences in the application, they could easier explain the status and the impediments. Other meeting participants were able to understand the projects status better than in the meetings without executable prototypes.

We also asked the release managers to introduce this technique in team and customer meetings. From team retrospective meetings in 2013, we received a lot of positive feedback about this possibility. If the meeting participants download the executable prototypes to the device in advance, this technique saves a lot of time and improves the communication and the understanding between different project participants, in particular between developers and the customer.

We consider these findings anecdotal evidence and conclude that the regular presentation of executable prototypes in Rugby improves the understanding of project participants, which supports hypothesis H1.5. The prerequisite for this finding is the existence of a fast and easy to use release mechanism that creates event based releases which can be used as executable prototypes.

7.3.6 Learning

Hypothesis H2.1 states that Rugby's workflows can be effectively taught by an instructor in university capstone courses and lectures within one semester. To evaluate H2.1, we conducted a quasi experiment (E3), in which we investigated whether the participants improved their technical skills and their soft skills as a result of taking our course. We looked at four categories *software engineering*, *usability engineering*, *configuration management* and *non technical skills*. First, we asked students about their skill improvements in software engineering with respect to requirements engineering, system design, modeling and programming.

On average, 79 % improved their requirements engineering skills (2011: 78 %, 2012: 80 %, 2013: 75 %, 2014: 81 %) and 72 % improved their system design skills (2011: 83 %, 2012: 78 %, 2013: 63 %, 2014: 72 %). The number of students who improved their modeling skills decreased by 25 % between 2011 and 2012 (see Figure 7.9), while at the same time the number of students who improved their programming skills increased by 28 % (see Figure 7.10). In 2012, it was the first time that we included an introduction to iOS programming before the course started. Due to the

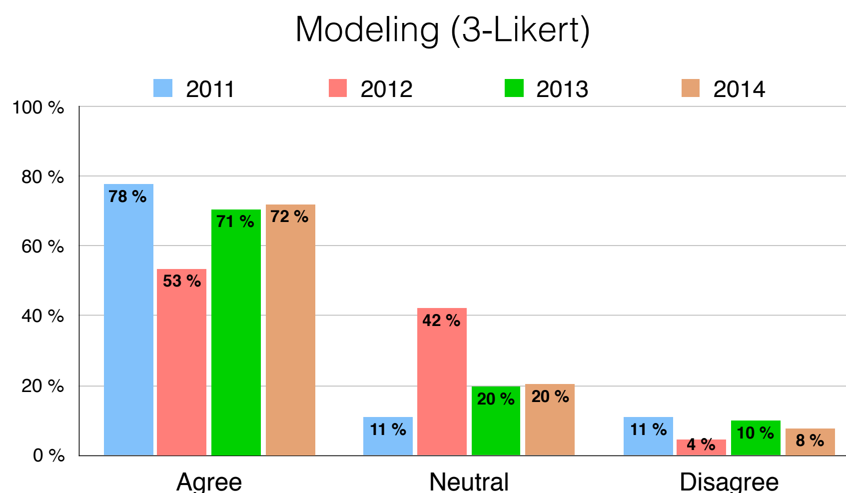


Figure 7.9: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *modeling* between 2011 and 2014

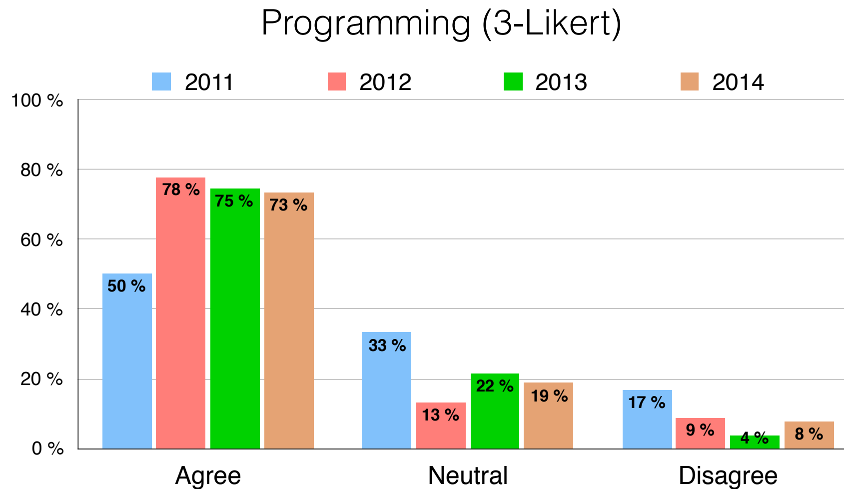


Figure 7.10: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *programming* between 2011 and 2014

higher emphasis on programming, we decreased the focus on formal modeling during the course. In 2013, the number of students who improved their modeling skills increased again because we emphasized user interface modeling and informal modeling techniques [DKA14].

We asked the students about their skills improvements in prototyping and user interface design. On average, 75 % of the students improved their prototyping skills (2011: 72 %, 2012: 80 %, 2013: 78 %, 2014: 69 %) and 65 % of the students improved their user interface design skills (2011: 72 %, 2012: 58 %, 2013: 63 %, 2014: 66 %). With respect to configuration management, we evaluated the students' skill improvements in version control and release management.

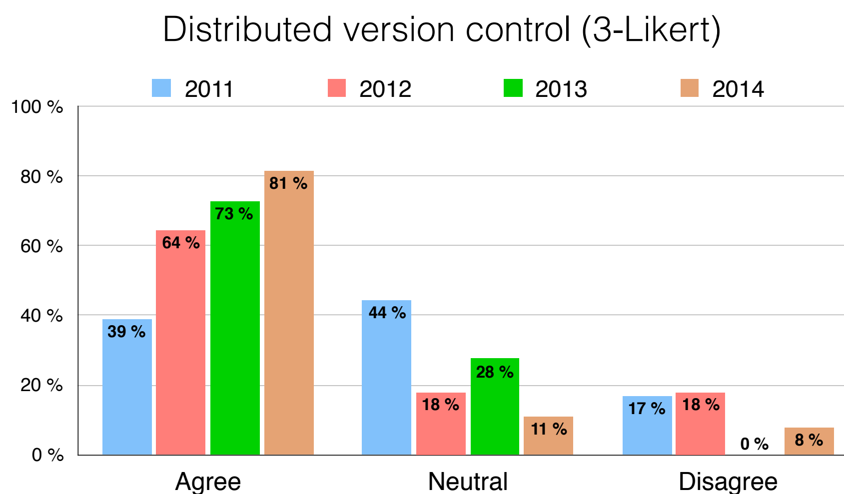


Figure 7.11: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *distributed version control* between 2011 and 2014

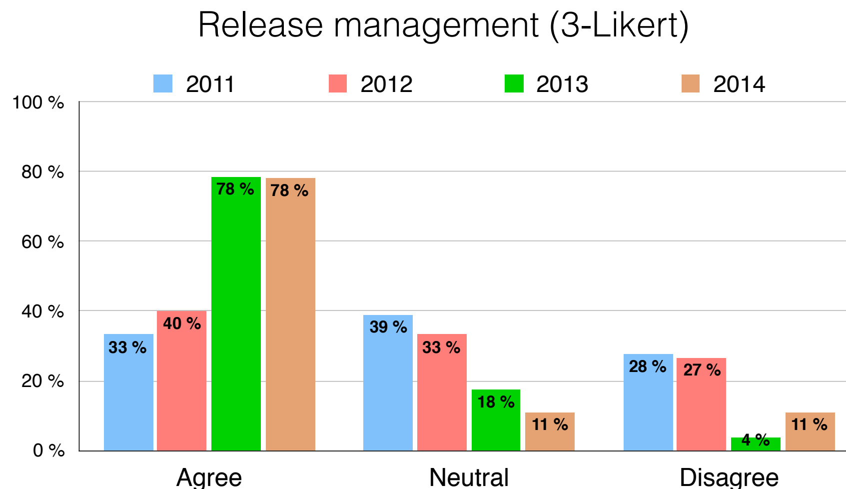


Figure 7.12: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *release management* between 2011 and 2014

Figure 7.11 shows a gradual increase of students who improved their version control skills from 2011 to 2014. The reason is the introduction of the branching model and stronger emphasis on dedicated code review workflows. Figure 7.12 shows the number of students who improved their release management skills doubling from 2012 to 2013. The reason is the early introduction of the continuous delivery workflow in the course.

We asked the students about their skill improvements with respect to communication, team work, presentation and demo management. Figure 7.13 and Figure 7.14 show that continuously more than 80 % improved their communication and team work skills in each course. For most students our course is their first experience working in large teams overcoming cultural differences and negotiating with a client. In addition,

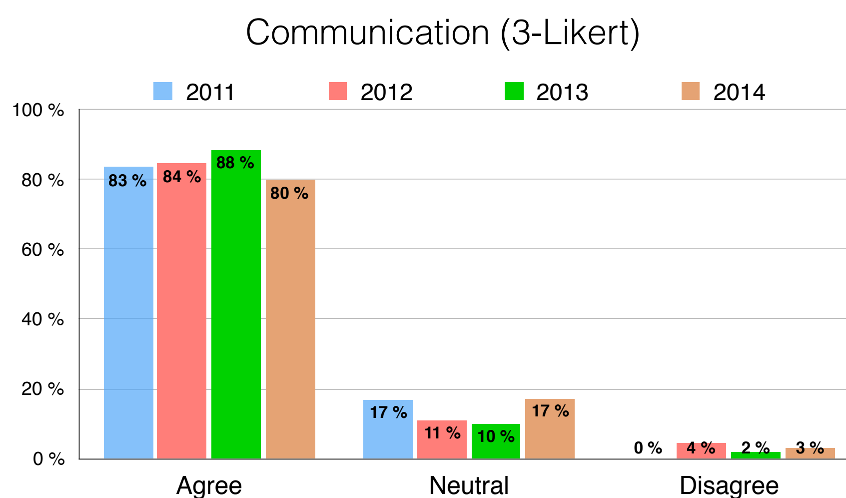


Figure 7.13: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *communication* between 2011 and 2014

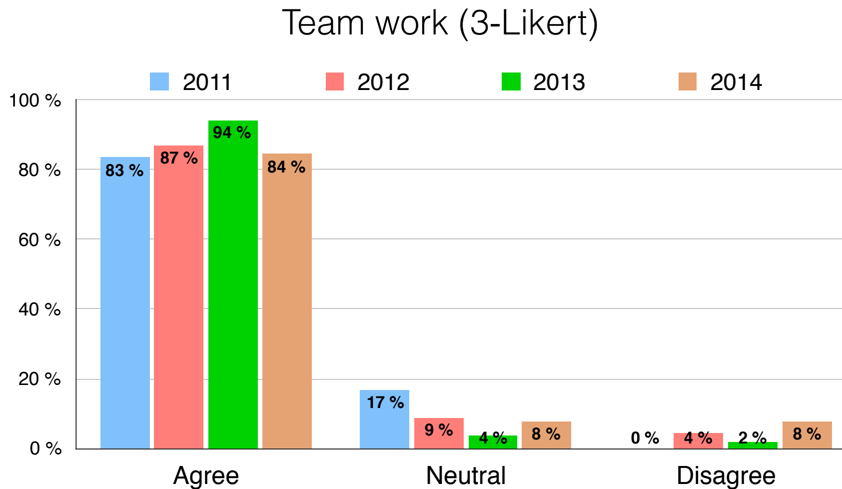


Figure 7.14: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *team work* between 2011 and 2014

they have to self organize each other in their team while the instructor makes sure that everybody contributes to the success of the project.

Figure 7.15 shows a gradual increase of the presentation skills from 56 % in 2011 to 83 % in 2014 and Figure 7.16 shows that the demo management improvement was on average 70 %. The possibility to watch the performance of their own presentation in the dry run and to get detailed feedback about technical aspects helped the students to improve their presentation skills. Our increased focus on demo management required more students to participate in the presentation. Up to 2011, it was normal that only one or two students performed the presentation, while in later years the whole team was involved. In addition, we film and stream the final presentations live into the

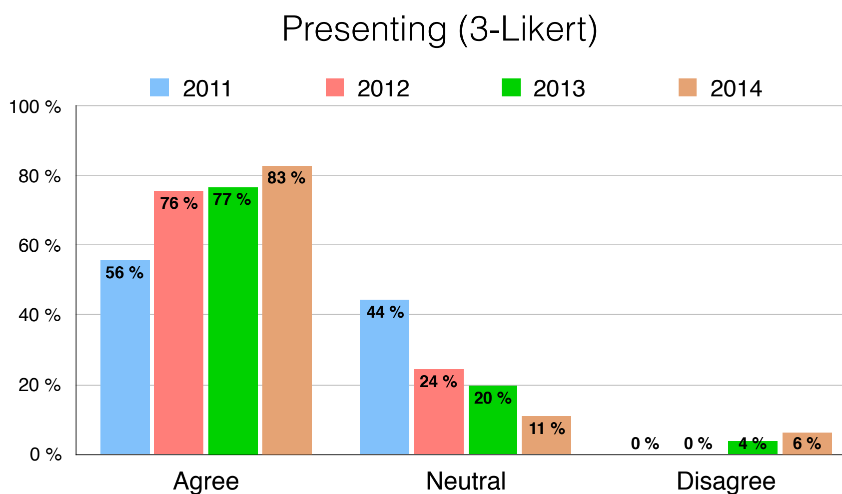


Figure 7.15: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *presenting* between 2011 and 2014

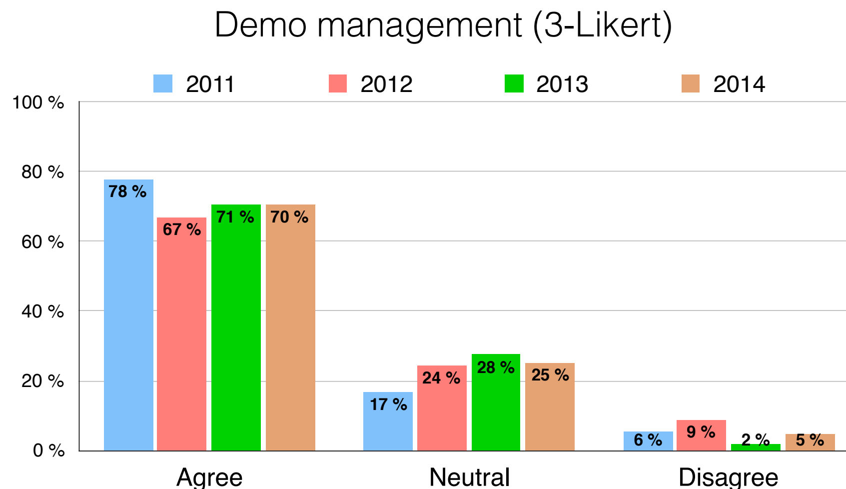


Figure 7.16: Answer of questionnaire participants (E3): Skill improvements in capstone courses in *demo management* between 2011 and 2014

Internet, so that global customers and the family members of students can watch the presentations. This increases the pressure to the students to improve their presentation skills in multiple dry runs. We observed that most of the teams asked for multiple internal dry runs to improve their presentations. The industry clients appreciate the improvements in the presentation, because they can use the videos to promote the idea in their company and for the further development of the application.

We found anecdotal evidence that students improve their technical and soft skills in capstone courses which use Rugby's process model. More students reported that they improved their skills, e.g. with respect to release management, in later years of the capstone course, when Rugby's workflows, e.g. release management, were used from the beginning of the course. We conclude that Rugby's workflows can be effectively taught by instructors in capstone courses, which supports hypothesis H2.1.

7.3.7 Scalability

Hypothesis H2.2 states that Rugby allows instructors to conduct exercises in large class rooms with more than 100 students. To evaluate H2.2, we investigated the introduction of five exercises in a large lecture with 423 registered students (E6). 223 students (53 %) participated in the questionnaire at the end of the course. As described in more detail in Section 6.2, the students first learned concepts in individual (tutorial based) exercises conducted in class. Then, they applied these concepts in team exercises. The questionnaire asked the students whether they participated in five individual exercises and whether they applied the following five techniques in their

team: agile methods, distributed version control, branch and merge management, continuous integration (including testing) and continuous delivery.

The questionnaire used a five point Likert type scale with the answers *strongly disagree*, *disagree*, *neutral*, *agree*, *strongly agree* to measure either negative, neutral or positive responses with respect to techniques learned in the five exercises:

1. **Improved**: In the exercises, I was able to improve my skills in the following techniques
2. **Confident**: I am confident to apply the following techniques in my next team project

In the following, we describe the results to these statements for students who participated in individual exercises **and** who applied the technique in their team. This means, for each technique we filtered the students who reported that they did not participate in the individual exercise or who reported that they did not apply the technique in their team. We did that, because answers of these students would distort the results.

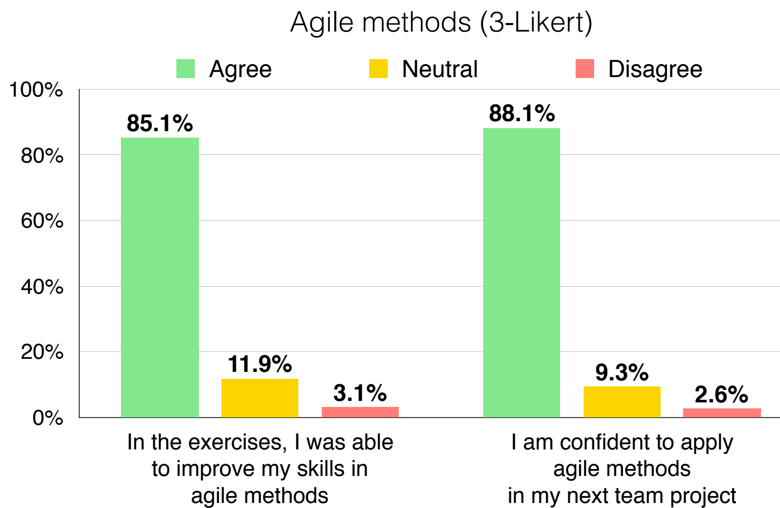


Figure 7.17: Answer of questionnaire participants (E6): improvements and confidence in *agile methods*

Figure 7.17 shows that 85 % of the students improved their skills in agile methods and that 88 % of the students are confident to apply agile methods in their next team project. This results confirms the strong focus of the exercises on agile methods and shows that students feel prepared for the management of their next agile project.

Figure 7.18 shows that 73 % of the students improved their skills in distributed version control. This number is lower because distributed version control is widely today in software engineering projects [BCSD14] and students also use it other projects besides the university. 86 % of the students are confident to apply distributed version control in their next team project.

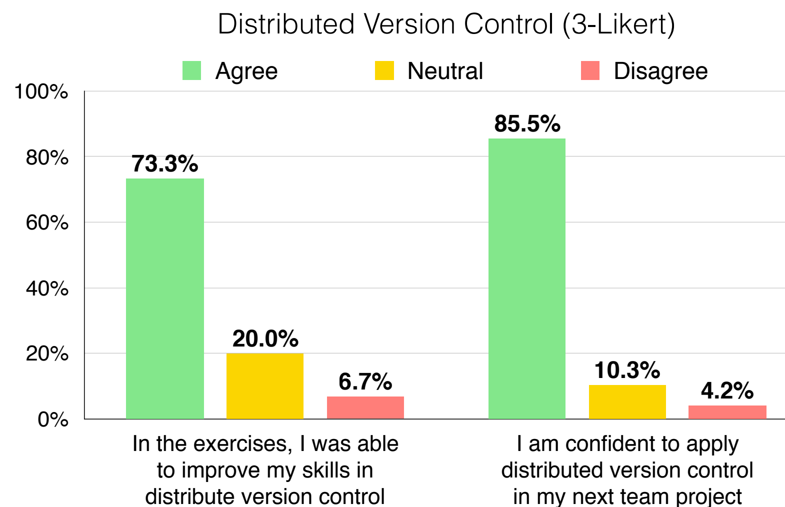


Figure 7.18: Answer of questionnaire participants (E6): improvements and confidence in *distributed version control*

Figure 7.19 shows that 78 % of the students improved their skills in branch and merge management which is part of Rugby's review management workflows. 87 % of the students are confident to apply it in their next team project. These results show that branch and merge management has become teachable. Students are able to handle parallel branches and can deal with merge conflicts.

Figure 7.20 shows that 76 % of the students improved their skills in continuous integration. The students could experience the benefits of immediate feedback about

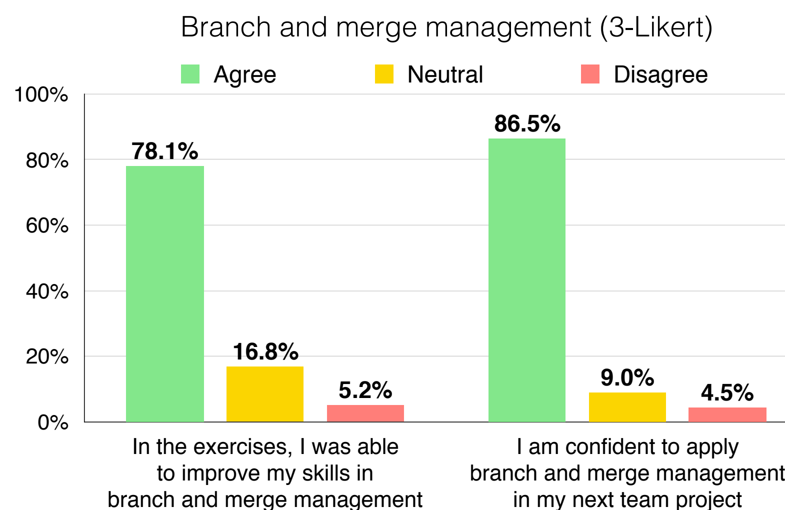


Figure 7.19: Answer of questionnaire participants (E6): improvements and confidence in *branch and merge management*

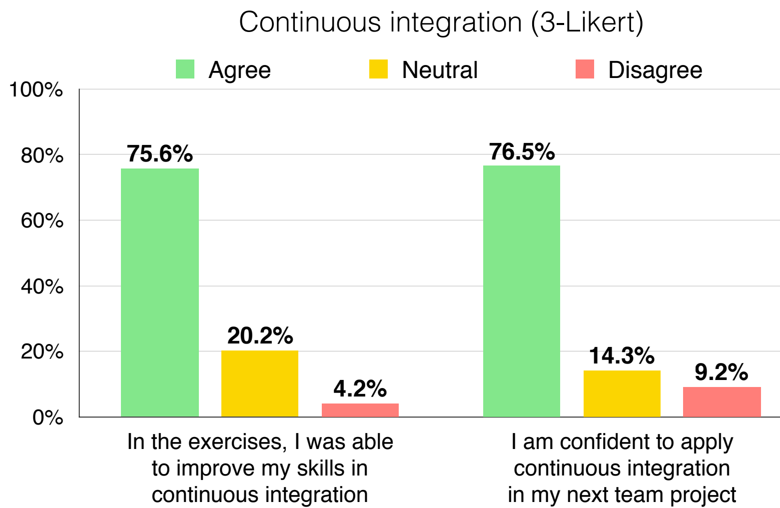


Figure 7.20: Answer of questionnaire participants (E6): improvements and confidence in *continuous integration*

integration and test failures after they committed their changes to the source code repository. 77 % feel confident to apply continuous integration in their next team project.

Figure 7.21 shows that 78 % of the students improved their skills in continuous delivery. In the release management exercise, they configured continuous delivery for a mobile application and applied it in their team project as well. 79 % of the students are confident to apply continuous delivery in their next team project.

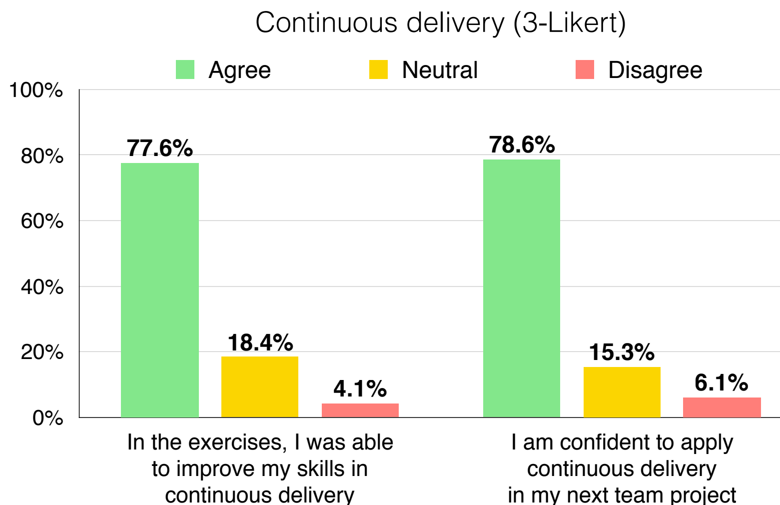


Figure 7.21: Answer of questionnaire participants (E6): improvements and confidence in *continuous delivery*

In Table 7.5, we summarize the evaluation results of the questionnaires for the five techniques introduced in the exercises and evaluated in the questionnaire. Table 7.5 includes the following three different filters:

(1) **Individual exercise:** We considered only students who reported that they participated in the individual exercise of the corresponding technique and filtered out students who reported that they did not participate in the individual exercise, independent whether the students applied the technique in their team project.
(2) **Team based exercise:** We considered only students who reported that they applied the concept in their team project and filtered out students who reported that they did not apply the technique in their team project, independent whether the students participated in the individual exercise.
(3) **Both exercises:** We considered only students who reported that they participated in individual exercises **and** who applied the technique in their team. We filtered out all other students. These are the same results as shown in Figure 7.17 - Figure 7.21.

Technique	**(1) Individual exercise**			**(2) Team based exercise**			**(3) Both exercises**		
	#	**Agree improved**	**Agree confident**	**#**	**Agree improved**	**Agree confident**	**#**	**Agree improved**	**Agree confident**
Agile methods	209	83.7 %	87.1 %	198	84.8 %	88.4 %	194	85.1 %	88.1 %
Distributed version control	195	71.3 %	83.6 %	172	73.3 %	86.0 %	165	73.3 %	85.5 %
Branch & merge management	197	75.6 %	81.7 %	162	75.9 %	85.8 %	155	78.1 %	86.5 %
Continuous integration	166	71.7 %	73.5 %	138	70.3 %	73.9 %	119	75.6 %	76.5 %
Continuous delivery	149	71.8 %	71.8 %	118	73.7 %	73.7 %	98	77.6 %	78.6 %

Table 7.5: Overview of subjective opinions of students about exercise improvements and confidence (E6) divided into students who participated in individual, team or both exercises: percentage of students who participated and (strongly) agree that they improved their skills, and percentage of students who participated and (strongly) agree that they are confident to apply the technique in their next project

In addition to the qualitative evaluation (E6), we also conducted a quantitative evaluation (E9) to find a correlation between exercise participation and the average grade of the students in the final exam. The students could receive up to 600 exercise points through the participation in quizzes, individual exercises and team based exercises. We grouped the 294 students, who participated in the exam, into five categories with equal distances describing their participation in the exercises. For instance, the first

category contains 75 students who obtained less than 20 % of the exercise points and the second category contains 66 students who obtained between 20 % and 40 % of the exercise points. Figure 7.22 shows that students with less exercise participation (i.e. with less exercise points) have worse grades than students with a higher participation.

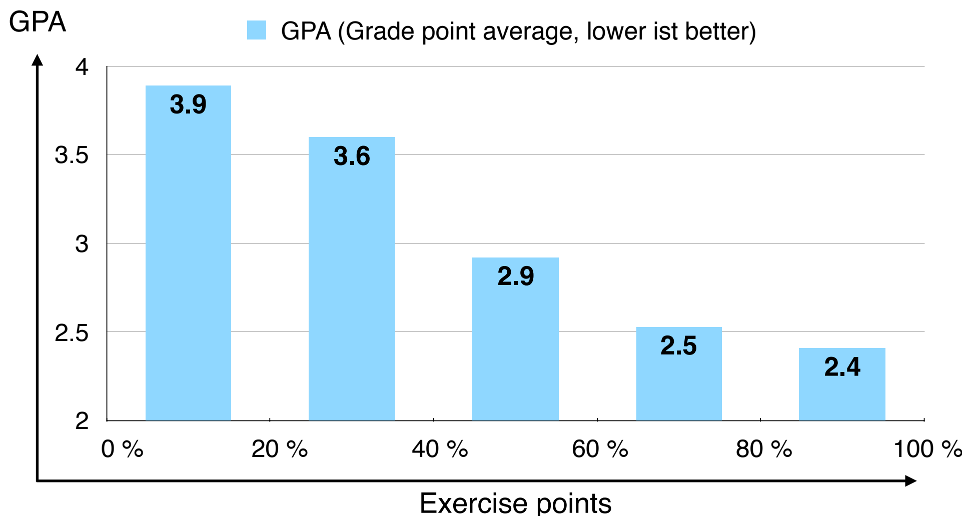


Figure 7.22: The GPA of the final exam grouped by students' exercise points (E9) shows correlation: students who successfully completed more exercises received more exercise points and scored better in the exam (grades vary between 1.0 and 5.0; a lower grade is better).

In fact, students with less than 20 % of the exercise points have a grade point average of 3.9 and students with more than 60 % have a grade point average of 2.5 or better[8]. We found the following correlation: A higher exercise participation leads to a better grade on average. Table 7.6 shows additional details of the correlation, such as how many students participated in the exam and how many students were in each exercise point category. The final exam included questions about the five techniques mentioned in Table 7.5. A better grade in the exam means, that the students showed a better understanding of these techniques. Therefore, we can conclude that the exercises improved the understanding of the students.

We identified a correlation between the exercise participation and the final exam results. Therefore, we conclude from these findings that Rugby's workflows can be effectively taught by instructors in capstone courses, which supports hypothesis H2.1. We can also conclude that instructors can conduct exercises in large class rooms with more than 100 students, which supports hypothesis H2.2.

[8]The grade point averages in Figure 7.22 and Table 7.6 do not include the bonus that the students could receive through exercise participation.

Exercise points (absolute)	**0 - 119**	**120 - 239**	**240 - 359**	**360 - 479**	**480 - 600**	**All**
Exercise points (relative)	0 % - 20 %	20 % - 40 %	40 % - 60 %	60 % - 80 %	80 % - 100 %	-
Number of students who took the exam	75	66	88	56	9	294
Number of students who passed	42	48	79	55	9	233
Number of students who failed	33	18	9	1	0	61
GPA without bonus (all students)	3.9	3.6	2.9	2.5	2.4	3.2

Table 7.6: Correlation between exercise participation and GPA in the final exam: students who successfully completed more exercises received more exercise points and scored better in the exam (grades vary between 1.0 and 5.0; a lower grade is better, a higher grade is worse).

7.4 Limitations

As is the case with many qualitative studies, ensuring the validity of findings is a challenging undertaking [Gol03]. In this section, we describe the limitations of our qualitative and quantitative evaluation[9].

E1 and E2: Questionnaires in the Capstone Courses in 2013 and 2014

There are threats to the validity in the methodology of evaluation E1 and E2 that we discuss briefly. First, we might have the problem of selection bias. Some teams used the workflows more frequently than others because of more experienced students. To alleviate selection bias we asked students in which team they worked. We have at least three responses in E1 from each team and at least five responses in E2 from each team, so the threat of selection bias is small. Additionally we observed the same results in the interviews where interviewees agreed with our findings.

A problem might be that participants gave answers which do not reflect their work practice, because they knew we would publish the results. We guaranteed the participants the complete anonymity and addressed this threat by that. We know that our findings might not be generalizable to industry projects because of the different setup at university. However, we think our course is similar to a project based organization in industry so that most results would remain valid. In addition, we could validate some results in an evaluation in industry (see E4). The amount of code lines changed in a

[9] To avoid repetition, we do not mention specific biases multiple times.

pull request is not shown. It does not reflect the size of the code changes as developers added frameworks, used automatic code formatting and refactored code.

Another limitation is that we cannot compute the number of response commits after the pull request was created to address review comments in the initial change set. It would be interesting to evaluate how many comments developers addressed in these response commits. We were not able to measure the code quality automatically and how much it improved in contrast to previous courses. We reviewed random samples and observed that it increased. Our evaluation mainly focused on the benefits through the process aspects of the workflow and it was not our aim to measure code quality in a quantitative manner.

E3: Questionnaire in the Capstone Courses in 2011 - 2014

In the quasi experiment E3, we did not use a control group within the same course; instead we did a formative evaluation where we compared the previous course without intervention with the successor course that used the intervention. Small differences in the organization of the courses could have influenced the results. Even though it is a quasi experiment, we think that the results are generalizable for other capstone courses. Another threat is that we used Likert type scales which may be subject to distortion.

Respondents may have avoided using extreme response categories (central tendency bias) and may have agreed with statements as presented (acquiescence bias). They may have tried to portray themselves or our course in a more favorable light (social desirability bias). As we designed the Likert type scale with balanced keying (i.e. an equal number of positive and negative statements), we obviated the problem of acquiescence bias, since acquiescence on positively keyed items will balance acquiescence on negatively keyed items. Our findings apply to a multi customer capstone 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 easily.

E4: Questionnaire in Industry in 2014

In the industry evaluation E4, we wanted to gain insight into the development process of mobile projects and measure the impact of our workflow. We thoughtfully worded each question to avoid ambiguity of leading questions. However, our results may be subject to the following limitations:

Factors such as duration of evaluation period, number of metrics, or level of detail may have influenced the reliability of our results. We can consider our findings anecdo-

tal evidence [RS11] for the impact of release and feedback management on mobile app development projects in a corporate environment. The number of projects and the variation of project characteristics is low in order to achieve generalizable results [PCL+04]. For example, we did not find a correlation between project size or complexity and any of the observed effects. However, consistent results across all eight interviewed projects are an indication that our results apply to other projects as well.

Projects participating in our personal interview may already have experience in agile methods and interest in automated integration and delivery [PCL+04]. This might impact our findings to the positive, though cultural acceptance is a prerequisite for successful agile projects [HF10]. Bias caused by an appreciation for continuous software engineering or our customized workflow in particular and positive results of previous studies [KABW14] may have influenced the wording of the questions. We chose project managers as interview participants with high expertise, which allows them to correct for ambiguity and added open ended questions to encourage full, meaningful answers to alleviate this threat.

E5: Personal Interviews in the Capstone Course in 2015

The interviews in our evaluation E5 were on a voluntary basis which leads to a potential problem with selection bias. The question arises if only the students who were convinced of the benefits of the review workflow were willing to participate in our study. We noted the team of each interviewee and had a diverse pool of participants that represent seven of the eleven teams from the WS 2014/15 capstone course. More than half of the review managers volunteered to join the study. To alleviate selection bias we also analyzed data quantitatively (see evaluation E8). Another limitation is that we were only able to interview 16 participants due to the limited number of volunteers paired with the substantial amount of time an interview took. Therefore, generalizations are difficult to make and the study rather aims to provide qualitative insights into how code reviews were conducted and how the students' regarded their value.

Given the nature of personal interviews person, the students might not entirely be forthcoming with negative information about their team members, the teaching assistants and the workflow itself. The lack of anonymity and the exposure of their person may have influenced their answers. To minimize this threat, we assured that their recounts would not be shared with any third party without being anonymized. The phrasing of some of the questions in the interview could be considered to lead the respondent to a certain answer. However, the questions were primarily used as manual to lead the discussion towards finding the opinion on the matter and its rationale. The

questions helped the interviewer and were not exposed to the participants so that the interview rather was a conversation instead of a question answer session.

E6: Questionnaire in the Lecture in 2015

In our evaluation E6 in the lecture in 2015 one threat to the validity is that the personal opinion of students does not reflect the real situation. Most students were beginners in the taught concepts. A student without previous knowledge in an area will definitely improve his knowledge, even if he only learns a limited amount of concepts. Beginners cannot estimate objectively about their real improvement and the confidence to apply a concept does not necessarily mean that the student is really able to apply it. Other positive effects of the lecture, such as the open atmosphere towards feedback, might have a positive influence on the evaluation result. Only if a student likes interactive exercises, this does not necessarily mean that he improves his skills. To alleviate these threats, we additionally evaluated the participation in the exercises quantitatively in a more objective manner, see E9.

E7: Quantitative Measurement in Capstone Courses between 2012 and 2014

In the quantitative measurement of release and feedback management numbers, we recognize the following limitations. More releases do not necessarily mean, the customer has a look at the release and more feedback does not necessarily mean the feedback is helpful or does contain good quality. However, the qualitative studies show that releases were helpful and that the quality of feedback was improved for automatic feedback reports. We were not able to distinguish whether developers and customers downloaded the releases. While both is helpful, we cannot say how often a customer actually downloaded the application. We also do not know how often the customers actually tried out the application after the downloaded it and how thoroughly they did it. However, we know from feedback after the courses and from the project leader that customers use this opportunity and that they like it.

E8: Quantitative Measurement in Capstone Courses between 2014 and 2015

In the quantitative measurement of review management numbers, we recognize the following threats to validity. More reviews do not necessarily improve the quality of source code. We know that some reviews were empty due to time pressure before milestones. In addition, more comments do not necessarily improve the communication because wrong and misleading comments might occur. However, the qualitative study showed

that these threats are low and that most reviews and comments led to quality improvements We could not quantitatively measure the actual code quality, because of missing tool support for Swift. Therefore, we cannot measure the quantitative improvements of the code quality, we only know that the quality improved from the qualitative study (see E2 and E5).

E9: Quantitative Measurement in Lecture between 2012 and 2014

In the quantitative measurement of the correlation between exercise participation and final exam grade, we recognize the following threats to validity. The exercise quality might not be the root cause for the improvement of the final exam grade. One other influence to the correlation might be the motivation of the students, because students who participate in exercises usually also have a higher motivation to learn the theory for the exam, or they might have more experience. We were not able to measure these variables or to exclude them. However, the qualitative evaluation E6 supports the results. Therefore we believe that these threats are low.

7.5 Summary

In our evaluation, we found anecdotal evidence that supports our two main hypotheses H1 and H2: Rugby allows to reduce the delay between development and usage in software projects and between teaching and exercising a concept in education. In addition, we found anecdotal evidence to support the dervied hypotheses. Only H1.3 is partially supported, because Rugby's feedback management workflow increases the quality of automatic feedback, however we did not find evidence that it also increases the quality of manual feedback. The summarized results of the findings of our evaluations in relation to the hypotheses are shown in Table 7.7.

ID	Hypothesis	Result
H1	Rugby allows to reduce the delay between development and usage in software projects.	Supported
H1.1	**Review**: Developers increase their code quality with Rugby's review management workflow.	Supported
H1.2	**Release**: Rugby's release management workflow reduces the effort required to create a release.	Supported
H1.3	**Feedback**: Rugby's feedback management workflow increases the quality of feedback.	Partially supported
H1.4	**Frequency**: The use of Rugby's process model increases the number of reviews, releases and feedback reports.	Supported
H1.5	**Understanding**: The presentation of event based releases as executable prototypes improves communication and understanding between project participants.	Supported
H2	Rugby allows to reduce the delay between lectures and exercises in education.	Supported
H2.1	**Learnability**: Rugby's workflows can be effectively taught by an instructor in university capstone courses and lectures within one semester.	Supported
H2.2	**Scalability**: Rugby enables instructors to scale exercises in large class rooms with more than 100 students.	Supported

Table 7.7: Results of the evaluations in relation to the stated hypotheses

Chapter 8

Conclusion

"To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science."

—Albert Einstein

With Rugby, we have shown that continuous software engineering has become possible. We established a software process model for continuous software engineering that is based on agile principles and iterative workflows. In this chapter, we summarize the contributions of this dissertation and propose future work.

8.1 Contributions

Rugby's process meta model describes software engineering as a set of continuously running processes called workflows describing activities in the software development lifecycle. Workflows subscribe to events, sleep until a subscribed event wakes up the workflow, perform work and then sleep again. We have demonstrated that linear, iterative and agile process models and Rugby's continuous process model itself can be instantiated with this meta model. Rugby's process model addresses three nonfunctional requirements: the process is tailorable, the workflows are customizable and the change model with the event hierarchy is extensible.

Rugby's process model integrates three workflows: review management, release management, and feedback management. Rugby's review management workflow includes a branching model and a quality gate that together prevent poorly written code from being integrated into the main codebase where it would be distributed to the whole team. Developers improve the code quality and the understanding of the system through peer reviews. Rugby's release management workflow includes continuous

integration and continuous delivery activities, reduces the effort required to create a release and allows to release a change in the source code in an easy, fast and robust way to the user. Event based releases allow developers to quickly obtain feedback from customers anytime in the project lifecycle, if they have clarification questions. Rugby's feedback management workflow includes a semi automatic approach to enable user involvement with context sensitive user feedback. It includes a voting and comment system to reduce the number of duplicates, to optimize the feedback's quality and to help in situations of conflicting feedback. It is important to reduce effort on both user and developer sides when composing, sending and analyzing feedback and when improving the software according to the feedback.

We developed a reference implementation for Rugby and demonstrated Rugby's applicability in three case studies in university and industry. In the first case study, we applied Rugby's workflows in 62 university capstone course projects between 2011 and 2015. We conducted formative evaluations to analyze the introduction of Rugby's workflows in the capstone courses. From 2011 to 2014, the number of students in capstone projects who have improved their skills in configuration and release management increased from 40 % to 80 %. Rugby led to 96 code reviews, 64 delivered releases, 136 downloads and 27 feedback reports on average per team in 2014.

Another contribution is the application of Rugby in a lecture environment based on experiential and blended learning. In the second case study, we used Rugby in a university course in 2015, which demonstrates Rugby's extensibility. A qualitative study showed that Rugby improves the communication between instructors and students. 77 % of the students who participated in the exercises are confident to apply continuous software engineering workflows in future projects. We identified a correlation between the exercise participation and the final exam results.

In the third case study, we applied Rugby to eight industry projects in 2014, which demonstrated Rugby's customizability for different project environments. The projects adapted Rugby's release and feedback management workflows to their needs. A qualitative analysis confirmed the results that we found in the capstone projects in university and showed that the delay between development and usage of software is reduced in industry as well. The professionals in the eight industry projects were able to reduce the time effort for integration and delivery from hours to minutes, while increasing the frequency of releases.

8.2 Future Work

We have identified topics that can be further improved and extended. A CASE (computer aided software engineering) tool that implements Rugby's process meta model could automize the adaptations of the process for a concrete ecosystem. The CASE tool would enable the visualization of process tailoring, workflow customization and process extension, and it would generate the project environment and the reference implementation for the adapted process. Further research is needed about the quality of manual feedback and the use of a semi automated feedback workflow, to answer questions about the effort a developer can save through attached usage context.

Another area of future work would be the formalization of Rugby's process model with first order logic. A challenge is the combination of knowledge that was generated through Rugby's continuous software engineering approach with rationale management. This would allow developers to continuously reflect user behavior and user feedback, and support development teams in their software evolution decisions. Developers could reflect about the rationale of decisions made in previous product increments and managers could use this knowledge for decision support.

Rugby already allows the exploration of multiple variations of requirements through feature branches. Development teams can deliver such variations through event based releases from these features branches to users and collect user feedback. This sets up a search space with multiple proposals. Currently, knowledge management is independent of this search space and the decision which proposal is chosen is not captured as rationale in the knowledge management repository. Therefore, this knowledge is not accessible in later iterations of the project.

A different research area is the further automatization of exercises in large class rooms through Rugby. This would allow to scale the exercises to large heterogeneous MOOCs (Massive Open Online Course), which are aimed at “unlimited” participation and open access via the Internet. Rugby allows to combine theory and exercises in small interactive units shortly after each other to improve the learning experience in classes. In a MOOC, content is delivered through videos, where the interactive feedback loop is missing. Rugby could allow the combination of theory and exercises in videos and still provide this feedback loop through immediate responses to students whether their exercise solution is correct or not.

We already validated aspects of Rugby in industry by customizing the release and feedback management workflows for its use in eight projects at Capgemini. The validation of additional aspects such as review management in industrial projects is an area for future work.

Appendix A

Terminology

This appendix chapter provides an alphabetically sorted summary of terminology and definitions used in this dissertation.

Beta tester. A person who tests the application in its target environment, typically before an official release to the whole user base. [BD09]

Blended learning. Educational approach that allows students to learn through delivery of content and instructions via computer-mediated activities, digital and online media. [GK04]

Code quality. Conformance to functional requirements and the system architecture, in addition to the usage of corresponding design patterns and code guidelines, while avoiding development anti patterns and code smells.

Code review. Manual assessment of source code by humans, mainly intended to identify defects and quality problems. [BBZJ14]

Continuous integration. A development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. [Fow06]

Continuous delivery. A development practice - based on continuous integration - in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time. [Che15]

Continuous deployment. A development practice - based on continuous delivery - in which every change that passes automated test is automatically deployed to production. [HF10]

Cooperative learning. Educational approach which aims to organize classroom activities into social learning experiences where students work in groups to complete tasks collectively towards a common goal. [J+91]

Defined process control. Defined process control requires that every piece of work be completely understood. Given a well defined set of inputs, the same outputs are generated every time. [SB02]

Empirical process control. Empirical process control allows to control complex processes, which cannot perfectly be defined and which would generate unrepeatable and unpredictable results through visibility, inspection and adaption. [SB02]

End user. A person who has an interaction with the computer based information system as a consumer or producer/consumer of information. [CK89]

Experiential learning. Educational approach in which educators engage with students in direct experience to increase knowledge, develop skills, and clarify values, to facilitate the process of learning from experience. [Kol84]

Formal review. Formal assessment of something with the intention of instituting change if necessary. [Ste10]

Informal review. Flexible assessment that is customized to the situation and that is conducted as needed.

Lead user. A person whose present needs will become general in the future and who benefits significantly by having those needs met. [Hip86]

Methodology. A collection of methods for solving a class of problems. [BD09]

Problem based learning. Educational approach to learn about a subject through the experience of problem solving. [BF98]

Process control. Deals with mechanisms for maintaining the output of a process in a specified and desired range. Control does not mean the process can be completely predicted (also compare defined process control and empirical process control).

Process customization. Adapting a process to operational needs on a smaller level through removing, modifying or adding specific activities to an existing workflow, without significantly deviating from the workflow model of the process.

Process extension. Adapting a process to operational needs through adding a new workflow or activity that cannot be described with existing elements of the process model.

Process tailoring. Adapting a process to operational needs on a higher level through removing, modifying or adding specific workflows, without significantly deviating from the process model.

Product Owner. A facade to all interested stakeholders, who defines the product by describing and prioritizing the most valuable requirements for the proposed system. [SB02]

Quality. Conformance to flexible specifications that respond to the changes of the user's needs, in addition to the usage of corresponding patterns to address non-functional requirements if applicable, while avoiding anti patterns.

Scrum Master. A facilitator, who resolves impediments in the team and who is responsible that the team follows the basic Scrum process. [SB02]

Software configuration management. A project function, which encompasses the disciplines and techniques of initiating, evaluating and controlling change to software products, with the goal to make technical and managerial activities more effective. [BD09]

Software development process. A set of software development activities performed toward a specific purpose.

Software evolution. The continual development and adaption of a software system to keep it up to date with the changing environment, to satisfy stakeholders and to remain at an acceptable level in a changing world. [LB85]

Software life cycle. All activities and work products necessary for the development of a software system. [BD09]

Software life cycle model. An abstract characterization of how software should be developed for the purpose of understanding, monitoring, or controlling it. [Sca01]

Software process model. An abstract representation of a process describing the way of working in a software project.

Software process metamodel. An abstraction representation of a process model.

Software project. An endeavor undertaken to meet unique goals and objectives to bring beneficial change or add value by developing a software system.

Software project management. An activity during which managers plan, budget, monitor, and control the development process. Project management ensures that constraints and project goals are met. [BD09]

User feedback. Comments, complaints or requests directly from users about the satisfaction or dissatisfaction with a software system used as an important resource for improving the software system.

User involvement. A systematic exchange of information between (prospective) users and developers aiming for a better understanding of user needs and a consequent improvement of the software. [Pag13]

Version control system. A system that records changes to a file or set of files over time so that specific versions can be recovered. Local, centralized and distributed version control systems exist. [Cha09]

Workflow. A thread of cohesive and mostly sequential activities performed by project participants that produce artifacts. [BD09]

Appendix B

Process Models

In this appendix chapter, we describe Scrum as a prominent instance of an agile process model and the Unified Process as a prominent instance of an iterative process model. Both play an important role for this dissertation, because the Rugby meta model allows the instantiation of both process models.

B.1 Scrum

Scrum [Sch95] is an agile process model [Ver15] and is based on the agile manifesto [BBVB+01] published 2001 to uncover "better ways of developing software by doing it and helping others do it" [BBVB+01]. Some of the participants formed the Agile Alliance[1] later in the same year which is now the main organization behind Scrum. As a response to defined process control, the manifesto focuses on these four items to be more important for agile processes:

> ***Individuals and interactions*** *over processes and tools*
> ***Working software*** *over comprehensive documentation*
> ***Customer collaboration*** *over contract negotiation*
> ***Responding to change*** *over following a plan*
>
> — Agile Manifesto [BBVB+01]

Preconditions for successful projects are communication and collaboration within the team. Tools can facilitate and improve the working process, but the team members

[1] http://www.agilealliance.org

(individuals) and the interactions between them (e.g. in meetings) are more important. Teams do focus on working software because that is what the customer normally prefers over a comprehensive documentation. Documentation is important but executable software in the target environment has higher priority.

Collaboration with the customer is essential and communication about changes in requirements and adaptions to meet the customers is more important than a contract that has to be negotiated. The idea of dynamic project management is also known as the Polynesian navigation principle where everything "begins with an objective instead of a plan" and experience and the occurrence of unexpected events determine further steps. [BD09]

Scrum follows an incremental process with short time boxed iterations called **Sprint**s. An informal overview is shown in Figure B.1. While the chosen requirements in a sprint are fixed in the *Sprint Backlog*, they can change between sprints in the *Product Backlog*. This gives the development team a fixed period without disturbance within the sprint to focus on the realization, but allows the team to respond to changes between sprints. Scrum maximizes risk management and transparency through frequent introspection.

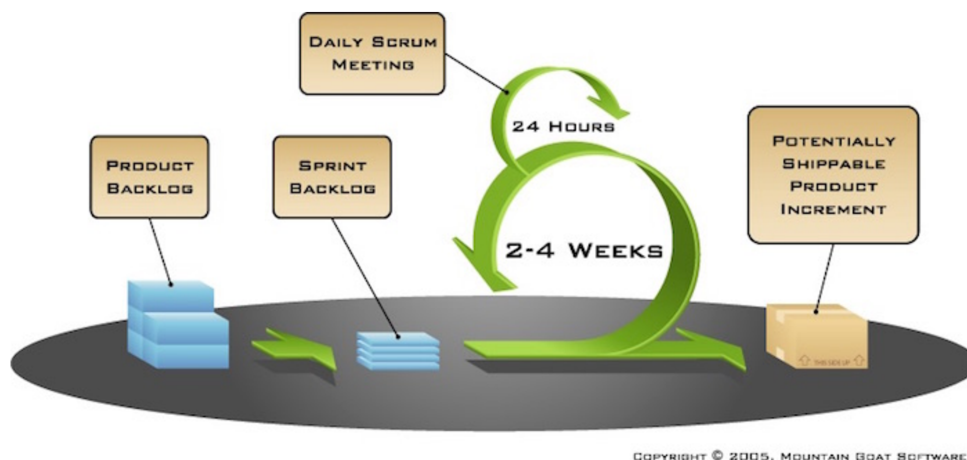


Figure B.1: Overview of the main Scrum activities and meetings [Mou05]

It defines three roles: Scrum Master, Product Owner and developer. A **Scrum Master** is different to a traditional project manager as he has no decision power. While he facilitates, resolves impediments and is responsible that the team follows the basic Scrum process, he does not assign tasks to **developer**s who form a self organizing and cross functional team that is responsible for realizing the product increment. A **Product Owner** is the facade to all interested stakeholders such as the customer, external managers and users. As the owner of the product he takes the role of the requirements engineer, defines the product by describing and prioritizing the most valuable require-

ments for the proposed system. His role is comparable to a product manager who is responsible for the results of the project. [SB02]
Scrum uses three main artifacts [Coh04]:

- **Product Backlog**: List of backlog items (requirements) for the whole product
- **Sprint Backlog**: List of backlog items (requirements and tasks) for one iteration
- **Potentially Shippable Product Increment**: Release to the Product Owner that contains all results of the current sprint

The main goal of a sprint is to create a potentially shippable product increment or short product increment, which refines previously realized requirements and includes additionally realized requirements. Only completely realized backlog items, that fully cover the system from a vertical perspective, are integrated into the product increment. The increment is used to obtain feedback that is addressed by adding new backlog items, changing or removing existing ones and reprioritizing items of the product backlog.
Scrum defines the following five main meetings [Coh04]:

- **Project Kickoff Meeting**: At the start of the project, everyone meets to create and prioritize the initial product backlog.
- **Sprint Planning Meeting**: At the start of each sprint, everyone meets to estimate and to create the sprint backlog
- **Daily Scrum Meeting**: Every day, the Scrum Master and the developers meet to share status since the last meeting, discuss impediments and promise work until the next meeting
- **Sprint Review Meeting**: At the end of each sprint, developers demonstrate realized backlog items in the product increment to the Product Owner and other participating stakeholders to obtain feedback.
- **Sprint Retrospective**: After the sprint review meeting, everyone meets to identify improvements in the process by discussing what went well and what went wrong in the last sprint.

Scrum defines additional artifacts such as burn down charts to track the progress of the team and task boards to visualize the status of tasks in the current sprint. While Scrum does not define a specific form of requirements, many teams use user stories to split the requirements into vertical items that can be implemented quickly [Coh09]. One technique to estimate the effort of backlog items is planning poker [Hau06], a team based technique based on expert estimates that usually uses Fibonacci numbers to estimate relative effort of backlog items. The idea of planning poker is to facilitate

initial discussion about backlog items: only if all developers come to an agreement for the estimate, the estimation of an item is finished [Coh06]. If different developers have varying opinions about the estimate, they have to exchange their arguments and find consensus.

B.2 Unified Process

The Unified Process [JBR98] is an iterative and incremental software development process framework that can be customized for organizations and projects. It divides the project in four different phases: inception, elaboration, construction and transition. Each phase can have multiple iterations and has a specific emphasis on different workflows as shown in Figure B.2. Workflows are also called process disciplines or engineering disciplines and include business modeling, requirements, analysis & design, implementation, test and deployment. The key process characteristics of the Unified Process are:

- **Iterative and Incremental**: Phases are divided into time boxed iterations which each result in an increment, a release including added or improved functionality.
- **Use Case Driven**: The development team describes use cases to explore functionality and content of requirements and to drive all development work, from requirements elicitation through analysis, design and code.
- **Architecture Centric**: The architecture is modeled upfront in the elaboration phase and the executable architecture baseline, a partial implementation of the system, validates the architecture and is the foundation for the rest of the project.
- **Risk Focused**: Addressing the most critical risks early in the project life cycle is required. Through the creation of an executable architecture baseline, important business and technical risks are identified, assessed and mitigated early.

The project starts with the relatively short **inception phase**, where the project team answers the question “Should we build the proposed system?” by establishing a business case for the project. Other typical goals include the definition of the project scope, boundary conditions and the outline of key requirements that influence the candidate architecture and the design tradeoffs. The team outlines a candidate architecture by creating initial versions of different models, identifies critical risks and determines when and how these risks will be addressed in the project. The milestone at the end of the inception phase is called life cycle objective. It is reached when the stakeholders agree on the scope of the propose system, when the candidate architecture addresses the

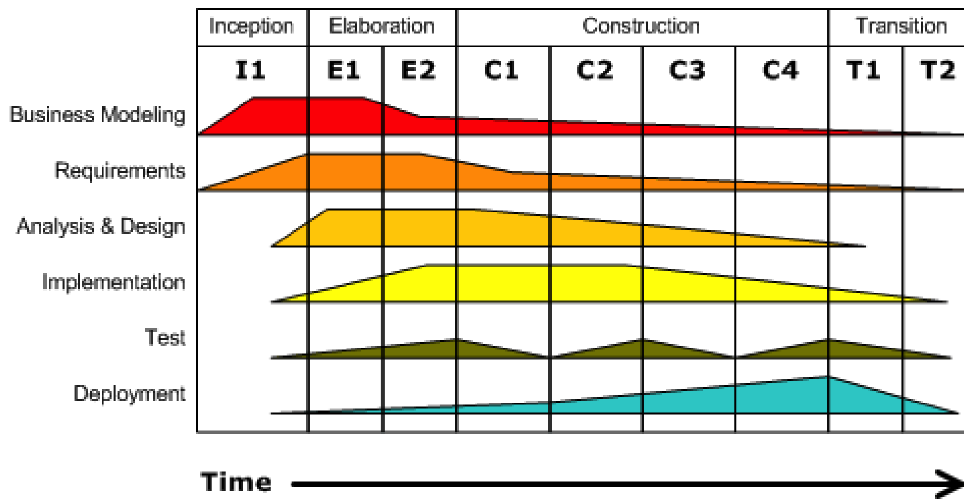


Figure B.2: Diagram illustrating how the relative emphasis of core workflow activities in the Unified Process change over the project duration [Kru04]

critical requirements and when the business case is valuable enough for continued development. [Sco02]

The **elaboration phase** addresses the question “Can we build the proposed system?”. The goal is to establish the ability to build the proposed system in the given constraints. Important tasks in this phase include the elicitation and analysis of remaining functional requirements as use cases and the transition of the candidate architecture into an executable architecture baseline, an internal release that validates the feasibility of the proposed system. Significant risks are assessed on an ongoing basis and the project plan is further detailed for the next phase. The milestone at the end of the elaboration phase is called life cycle architecture. It indicates that the most important functional requirements for the new system have been analyzed in the use case model, the architecture baseline is available as a solid foundation for ongoing development and the business case has green lights to proceed to the next phase. [Sco02]

In the **construction phase**, which is typically the largest phase of the project, the team actually implements the proposed system based on the foundation laid in the elaboration phase. It answers the question “Are we building it”. Functionality is realized incrementally in a series of short and time boxed iterations so that the proposed system is capable of operating successfully in beta environments. The team creates executable release in each iteration making sure the realized functionality of the system is always available in executable form. The milestone at the end of the construction phase is called initial operational capability. The project reaches this milestone if a fully operational system was successfully tested in beta environments. [Sco02]

The final phase in the Unified Process is called **transition phase** and answers the question “Have we delivered it?”. The fully functional system is rolled out to customers and eventually handed over to operations. The development team modifies the system only slightly to correct previously unidentified problems and deviations and corrects additional defects found in operation. The last milestone is the product release which marks the end of the project. [Sco02]

There are multiple refinements and variations of the Unified Process such as the Rationale Unified Process [Kru04], a commercial product of the company Rational Software which belongs to IBM. The Rational Unified Process includes three additional workflows as supporting disciplines for environment, project management, configuration and change management. It is a tailorable process to guide development including tools that automate the application of the process and services that accelerate the adoption of processes and tools [Kru04]. The Enterprise Unified Process [ANV05] is an extension of the Rational Unified Process and introduces eight additional enterprise disciplines: operations and support, enterprise business modeling, portfolio management, enterprise architecture, strategic reuse, people management, enterprise administration and software process improvement. It also adds two additional phases for production and retirement of software [ANV05]. We describe two other variations of the Unified Process, the Agile Unified Process and Disciplined Agile Delivery, as related work in Section 4.6.

Appendix C

Rugby’s Full Change Model

Figure C.1 shows Rugby’s full change model that includes all events used in this dissertation in the following figures:

- Dynamic view of the Waterfall process model: Figure 3.10
- Dynamic view of the Unified Process model: Figure 3.13
- Dynamic view of the Scrum process model: Figure 3.16
- Dynamic view of the *Develop Backlog Item* activity in the Scrum process model: Figure 3.16
- Dynamic view of the Rugby process model: Figure 4.9
- Dynamic view of the parallel workflows in the Rugby process model: Figure 4.11

Rugby’s change model is extensible, which means that events can added, changed and removed through process tailoring and workflow customization or through the use of Rugby into other domains.

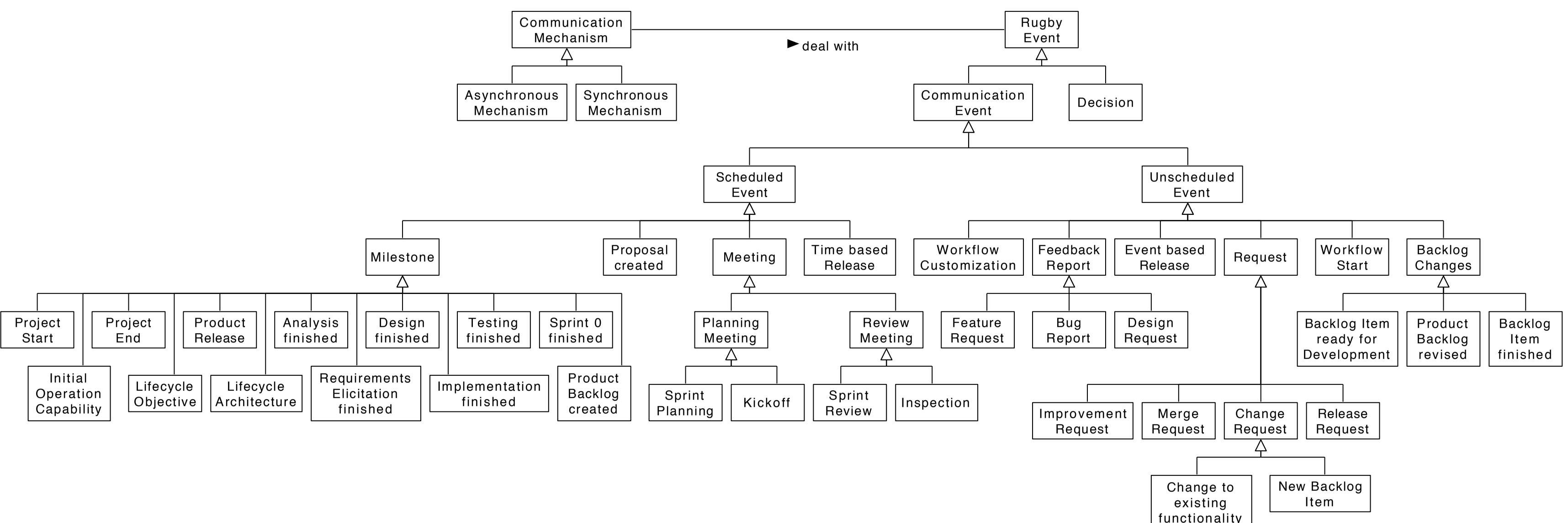


Figure C.1: Rugby's full change model with all events

List of Figures

1.1 Interaction between review, integration, test, delivery and feedback loops in Rugby . . . 7

2.1 Git flow branching model (adapted from [Dri10]) . . . 16
2.2 Deployment process with integration stage, test stage(s) and delivery stage (adapted from [HF10]) . . . 19
2.3 Review taxonomy (adapted from [CBDT06]) . . . 22
2.4 Activities in informal code reviews (adapted from [CBDT06]) . . . 23
2.5 Circular model of feedback in software evolution (adapted from [Sch11]) 24
2.6 Exemplary taxonomy of roles in the user feedback process (non exhaustive, adapted from [DKAB16]) . . . 25

3.1 Rugby's process meta model in the meta object facility layers . . . 29
3.2 Overview of Rugby's process meta model . . . 31
3.3 Static view of Rugby's process meta model as UML class diagram describing the core concepts and their relationships . . . 32
3.4 Simplified version of Rugby's Change Model including an event taxonomy for scheduled and unscheduled events . . . 34
3.5 Dynamic view of Rugby's process meta model as UML activity diagram describing the core activities and their control flow . . . 35
3.6 Generic example for the communication between multiple workflows through Rugby events . . . 35
3.7 The lifecycle model of a Rugby workflow (UML State machine) . . . 36
3.8 Exemplary instantiation of Rugby's work queue (adapted from QNX Microkernel's process priorities [Hil92]) . . . 37
3.9 Static view of the Waterfall process model with a UML class diagram as an example of a linear instantiation of Rugby's process meta model (adapted from [Roy70]) . . . 38

3.10 Dynamic view of the Waterfall process model as an example of a linear instantiation of Rugby's process meta model 39
3.11 Lifecycle of the Waterfall process as view of its dynamic model 39
3.12 Static view of the Unified Process model with a UML class diagram as an example of an iterative instantiation of Rugby's process meta model (adapted from [JBR98]) 40
3.13 Dynamic view of the Unified Process model with a UML class diagram as an example of an iterative instantiation of Rugby's process meta model (adapted from [JBR98]) 41
3.14 Lifecycle of the Unified Process as view of its dynamic model 42
3.15 Static view of the Scrum process model with a UML class diagram as an example of an agile instantiation of Rugby's process meta model (adapted from [SB02]) 43
3.16 Dynamic view of the Scrum process model as an example of an agile instantiation of Rugby's process meta model 44
3.17 Details of the activity *Develop Backlog Item* in the dynamic view of the Scrum process model 44
3.18 Lifecycle of the Scrum process as view of its dynamic model 45

4.1 Shared understanding between developers and users through releasing executable prototypes and obtaining feedback 48
4.2 Top level design of Rugby's ecosystem 48
4.3 Team roles in Rugby (UML class diagram taxonomy) 56
4.4 Rugby's high level use case model 57
4.5 Rugby's review management use case model 57
4.6 Rugby's release management use case model 58
4.7 Rugby's feedback management use case model 59
4.8 Static view of Rugby's process model 60
4.9 Dynamic view of Rugby's process model 62
4.10 Dynamic view of Rugby's process model: Details of the activity Develop Backlog Item 63
4.11 Dynamic view of the synchronization of parallel Rugby's workflows through change events 64
4.12 Agile Unified Process Timeline with the distinction between development releases and production releases (adapted from [CPP10]) 65
4.13 Disciplined Agile Delivery Lifecycle (adapted from [AL12]) 66

5.1 Rugby's lifecycle as view of its dynamic model 67

5.2 Rugby's review model 70
5.3 Rugby's branching model 70
5.4 Overview of the review workflow with developer and reviewer 71
5.5 Workflow for the preparation activity in branch based code reviews (adapted from [KBB16]) 73
5.6 Workflow for the examination activity in branch based code reviews (adapted from [KBB16]) 74
5.7 Workflow for the rework activity in branch based code reviews (adapted from [KBB16]) 75
5.8 Workflow for the integration activity in branch based code reviews (adapted from [KBB16]) 76
5.9 Rugby's release model 78
5.10 High level release management workflow with roles, services and transitions (adapted from [KA14]) 79
5.11 Event based delivery in the context of the branching model with four examples for releases (adapted from [KA14]) 80
5.12 Rugby's feedback model shows concepts of the feedback workflow and their relations in a UML class diagram - colors highlight related concepts 84
5.13 Rugby's feedback management workflow (adapted from [KABW14]) . . 86
5.14 Three approaches for feedback provision (adapted from [Sch11]) 87

6.1 Environments of the capstone course including Rugby's ecosystem and a management environment or the general course organization (adapted from [BKA15]) 95
6.2 Organizational chart showing the project based organization of the capstone course with project teams and functional teams (adapted from [BKA15]) 98
6.3 Wallboard showing whether the current version of the software is releasable or not 103
6.4 Mapping between the deployment process and stages in Bamboo . . . 109
6.5 Extended and customizable release management workflows with colors to indicate activities (adapted from [KKP+15]) 114

7.1 Answers in personal interviews (E5): The developers' view on motivations and benefits of code reviews 127
7.2 Answer of questionnaire participants (E1): Rugby's branching model . . 128
7.3 Answer of questionnaire participants (E2): branch and merge management problems 128

7.4 Answer of questionnaire participants (E2) about review management . . 129
7.5 Answer of questionnaire participants (E1 and E2): future usage of Rugby workflows 130
7.6 Answers in questionnaire (E4): Utilization of feedback channels and frequency of feedback collection 131
7.7 Answers in questionnaire (E4): Perceived quality of feedback across feedback channel 132
7.8 Answer of questionnaire participants (E1): Rugby's release management workflow 133
7.9 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *modeling* between 2011 and 2014 137
7.10 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *programming* between 2011 and 2014 138
7.11 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *distributed version control* between 2011 and 2014 . . 138
7.12 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *release management* between 2011 and 2014 139
7.13 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *communication* between 2011 and 2014 139
7.14 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *team work* between 2011 and 2014 140
7.15 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *presenting* between 2011 and 2014 140
7.16 Answer of questionnaire participants (E3): Skill improvements in capstone courses in *demo management* between 2011 and 2014 141
7.17 Answer of questionnaire participants (E6): improvements and confidence in *agile methods* 142
7.18 Answer of questionnaire participants (E6): improvements and confidence in *distributed version control* 143
7.19 Answer of questionnaire participants (E6): improvements and confidence in *branch and merge management* 143
7.20 Answer of questionnaire participants (E6): improvements and confidence in *continuous integration* 144
7.21 Answer of questionnaire participants (E6): improvements and confidence in *continuous delivery* 144
7.22 GPA of the final exam grouped by students' exercise points (E9) 146

B.1 Overview of the main Scrum activities and meetings [Mou05] 161
B.2 Diagram illustrating how the relative emphasis of core workflow activities in the Unified Process change over the project duration [Kru04] 164

C.1 Rugby's full change model with all events 167

List of Tables

6.1 Overview of the case studies 90
6.2 Number of participants in the multi customer project courses between 2011 and 2015 92
6.3 Interventions in the multi customer capstone courses between 2011 and 2014 93
6.4 Overview of the used tools in the capstone course since 2013 94
6.5 Typical schedule for course wide meetings in the multi customer capstone course 104
6.6 Lecture and exercise schedule of *Software Engineering II: Project Organization and Management* in 2015 106
6.7 Individual exercises and the corresponding bonus points for the lecture *Software Engineering II: Project Organization and Management* in 2015 107
6.8 Team exercises and the corresponding exercise points for the lecture *Software Engineering II: Project Organization and Management* used in 2015 110

7.1 Overview of the formative and qualitative evaluations between 2013 and 2015 119
7.2 Overview of the formative and quantitative evaluations between 2012 and 2015 120
7.3 Measurements of average use of version control, release management and feedback management workflows per team in capstone courses (E7) 134
7.4 Measurements of average number or reviews, comments and commits per team in capstone courses (E8) 135
7.5 Overview of subjective opinions of students about exercise improvements and confidence (E6) 145
7.6 Correlation between exercise participation and GPA of students in final exams (E9) 147
7.7 Results of the evaluations in relation to the stated hypotheses 152

Bibliography

[AL12] Scott Ambler and Mark Lines. *Disciplined agile delivery: A practitioner's guide to agile software delivery in the enterprise*. IBM, 2012.

[Ala02] Ian Alam. An exploratory investigation of user involvement in new service development. *Journal of the Academy of Marketing Science*, 30(3):250–261, 2002.

[And10] David Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.

[ANV05] Scott Ambler, John Nalbone, and Michael Vizdos. *Enterprise unified process, the: extending the rational unified process*. Prentice Hall, 2005.

[BA04] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley, 2004.

[BAKHE03] Rachel Ben-Ari, Ronit Krole, and Dov Har-Even. Differential effects of simple frontal versus complex teaching strategy on teachers' stress, burnout, and satisfaction. *International Journal of Stress Management*, 2003.

[Bas96] Victor Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering*, pages 442–449. IEEE, 1996.

[BB13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of ICSE*, pages 712–721. IEEE, 2013.

[BBVB+01] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. *The Agile Alliance*, 2001.

[BBZJ14] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.

[BCS14] David Boud, Ruth Cohen, and Jane Sampson. *Peer learning in higher education: Learning from and with each other*. Routledge, 2014.

[BCSD14] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How Do Centralized and Distributed Version Control Systems Impact Software Changes? In *Proceedings of the 36th International Conference on Software Engineering*, pages 322–333, 2014.

[BD93] Adolf Bröhl and Wolfgang Dröschel. *Das V-Modell - Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Oldenburg, 1993.

[BD09] Bernd Bruegge and Allen Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 3rd edition, 2009.

[BF98] David Boud and Grahame Feletti. *The challenge of problem-based learning*. Psychology Press, 1998.

[BF14] Pierre Bourque and Richard Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society Press, 2014.

[BKA15] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education*, 15(4):17:1–17:31, 2015.

[BKW12] Bernd Bruegge, Stephan Krusche, and Martin Wagner. Teaching Tornado: from communication models to releases. In *Proceedings of the 8th edition of the Educators' Symposium*, pages 5–12. ACM, 2012.

[Blu92] Bruce Blum. *Software engineering: a holistic view*. Oxford University Press, 1992.

[BMMM98] William Brown, Raphael Malveau, Hays McCormick, III, and Thomas Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[Boe88] Barry Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[Boe00] Barry Boehm. Requirements that handle ikiwisi, cots, and rapid change. *Computer*, 33(7):99–102, 2000.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Cummings, 1991.

[Bos14] Jan Bosch. *Continuous Software Engineering*. Springer, 2014.

[BR05] Manfred Broy and Andreas Rausch. Das neue v-modell® xt. *Informatik-Spektrum*, 28(3):220–229, 2005.

[BRS09] Bernd Bruegge, Maximilian Reiss, and Jennifer Schiller. Agile principles in academic education: A case study. In *Sixth International Conference on Information Technology: New Generations*, pages 1684–1686. IEEE, 2009.

[Bun92] Bundesamt für Wehrtechnik und Beschaffung. *Software Development Standard for the German Federal Armed Forces, V-Model, Software Lifecycle Process Model*, BWB General Directive 250 edition, 1992.

[BVGW10] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs. In *Proceedings of the ACM conference on Computer supported cooperative work*, page 291. ACM, 2010.

[Bö01] Jürgen Börstler. Experience with work-product oriented software development projects. *Computer Science Education*, 11(2):111–133, 2001.

[Cav95] Angèle Cavaye. User participation in system development revisited. *Information & Management*, 28(5):311–323, 1995.

[CBDT06] Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. *Best kept secrets of peer code review*. Smart Bear, 2006.

[CBH91] Allan Collins, John Seely Brown, and Ann Holum. Cognitive apprenticeship: Making thinking visible. *American educator*, 1991.

[CF04] Kieran Conboy and Brian Fitzgerald. Toward a conceptual framework of agile methods: a study of agility in different disciplines. In *Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research*, pages 37–44. ACM, 2004.

[CGB+02] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[Cha09] Scott Chacon. *Pro git*. Apress, 2009.

[Che15] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *Software, IEEE*, 32(2):50–54, 2015.

[CK89] William Cotterman and Kuldeep Kumar. User cube: a taxonomy of end users. *Communications of the ACM*, 32(11):1313–1320, 1989.

[CLB03] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews: The state of the practice. *IEEE software*, 20(6):46–51, 2003.

[CMPP08] Maria Francesca Costabile, Piero Mussio, Loredana Parasiliti Provenza, and Antonio Piccinno. End users as unwitting software developers. In *Proceedings of the 4th international workshop on End-user software engineering*, pages 6–10. ACM, 2008.

[COB06] O. Creighton, M. Ott, and B. Bruegge. Software cinema-video-based requirements engineering. In *Proceedings of the 14th International Conference on Requirements Engineering*, pages 109–118. IEEE, 2006.

[Coh04] Mike Cohn. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[Coh06] Mike Cohn. *Agile estimating and planning*. Prentice Hall, 2006.

[Coh09] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison Wesley, 2009.

[CPP10] Ioannis Christou, Stavros Ponis, and Eleni Palaiologou. Using the agile unified process in banking. *Software, IEEE*, 27(3):72–79, 2010.

[Cro80] Philip Crosby. *Quality is free: The art of making quality certain*. Signet, 1980.

[CSA15] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21–31, 2015.

[CW00] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–247, 2000.

[Dam96] Leela Damodaran. User involvement in the systems design process - a practical guide for users. *Behaviour and Information Technology*, 15:363–377, 1996.

[Dep88] Department of Defense. *Military standard: defense system software development*, 1988.

[DKA14] Dora Dzvonyar, Stephan Krusche, and Lukas Alperowitz. Real projects with informal models. In *Proceedings of the 10th Educators' Symposium*, 2014.

[DKAB16] Dora Dzvonyar, Stephan Krusche, Rana Alkadhi, and Bernd Bruegge. Context-aware user feedback in continuous software evolution. In *Proceedings of the 1st International Workshop on Continuous Software Evolution and Delivery*. IEEE/ACM, 2016.

[DMG07] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson, 2007.

[DMSW98] David Duffy, Cooper McDonald, Olivier Schueller, and George Whitesides. Rapid prototyping of microfluidic systems in poly (dimethylsiloxane). *Analytical chemistry*, 70(23):4974–4984, 1998.

[DPL15] Andrej Dyck, Ralf Penners, and Horst Lichter. Towards definitions for release engineering and devops. In *Proceedings of the Third International Workshop on Release Engineering*, pages 3–3. IEEE, 2015.

[Dri10] Vincent Driessen. A successful git branching model, 2010. Retrieved January 08, 2016 from http://nvie.com/posts/a-successful-git-branching-model.

[DS76] Michael Doyle and David Straus. *How to make meetings work*. Jove Books, 1976.

[DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[Ere03] Justin Erenkrantz. Release management within open source projects. *Proceedings of the 3rd Open Source Software Development Workshop*, pages 51–55, 2003.

[Fag76] Michael Fagan. Design and code inspections to reduce errors in program development. *IBM Journal of Research and Development*, 15(3):182, 1976.

[Fag86] Michael Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[FCC13] Davide Falessi, Giovanni Cantone, and Gerardo Canfora. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. *IEEE Transactions on Software Engineering*, 39(1):18–44, 2013.

[Fei02] Armand Feigenbaum. *Total quality management*. Wiley, 2002.

[FFB13] Dror Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, pages 8–17, 2013.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson, 1999.

[Fow01] Martin Fowler. The new methodology. *Wuhan University Journal of Natural Sciences*, 6(1-2):12–24, 2001.

[Fow06] Martin Fowler. Continuous Integration, 2006. Retrieved January 08, 2016 from http://martinfowler.com/articles/continuousIntegration.html.

[Fow11] Martin Fowler. Frequency Reduces Difficulty, 2011. Retrieved January 08, 2016 fromhttp://www.martinfowler.com/bliki/FrequencyReducesDifficulty.html.

[FUP11] Roger Fisher, William Ury, and Bruce Patton. *Getting to yes: Negotiating agreement without giving in*. Penguin, 2011.

[GAB15] Emitza Guzman, Omar Aly, and Bernd Bruegge. Retrieving diverse opinions from app reviews. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–10. IEEE, 2015.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson, 1994.

[GK04] Randy Garrison and Heather Kanuka. Blended learning: Uncovering its transformative potential in higher education. *The internet and higher education*, 2004.

[Gla64] Thomas Gladwin. Culture and logical process. In Goodenough W., editor, *Exlorations in Cultural Anthropology: Essays Presented to George Peter Murdock*. McGraw-Hill, 1964.

[GM14] Emitza Guzman and Walid Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *22nd International Requirements Engineering Conference*, pages 153–162. IEEE, 2014.

[Gol03] Nahid Golafshani. Understanding reliability and validity in qualitative research. *The qualitative report*, 8(4):597–606, 2003.

[GQ95] Mark Ginsberg and Lauren Quinn. Process tailoring and the the software capability maturity model. Technical Report CMU/SEI-94-TR-024, Carnegie Mellon University, 1995.

[Gru91] Jonathan Grudin. Systematic Sources of Suboptimal Interface Design in Large Product Development Organizations. *Human-Computer Interaction*, 6(2):147–196, 1991.

[Han93] Charles Handy. *Understanding organizations: managing differentiation and integration*. Oxford University Press, 1993.

[Hau06] Nils Haugen. An empirical study of using planning poker for user story estimation. In *Agile Conference*. IEEE, 2006.

[HF10] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson, 2010.

[Hig02] Jim Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley, 1st edition, 2002.

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[Hip86] Eric von Hippel. Lead users: a source of novel product concepts. *Management science*, 32(7):791–805, 1986.

[Hoa78] Tony Hoare. Communicating sequential processes. *Communactions of the ACM*, 21(8), 1978.

[Hol05] Karen Holtzblatt. Designing for the mobile device: Experiences, challenges, and methods. *Communications of the ACM*, 48(7):32–35, 2005.

[HR00] David Hilbert and David Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, 2000.

[HT00] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley, 2000.

[Hum11] Jez Humble. Devops: A software revolution in the making? *Cutter IT Journal*, 24(8), 2011.

[Ins88] Institute of Electrical and Electronics Engineers. *Guide to Software Configuration Management (Standard 1042-1987)*, 1988.

[Ins08] Institute of Electrical and Electronics Engineers. *Standard for Software Reviews and Audits (Standard 1028-2008)*, 2008.

[J+91] David Johnson et al. *Cooperative Learning: Increasing College Faculty Instructional Productivity. ASHE-ERIC Higher Education Report*. ERIC, 1991.

[JBF13] Lynette Johns-Boast and Shayne Flint. Simulating industry: An innovative software engineering capstone design course. In *Frontiers in Education Conference*, pages 1782–1788. IEEE, 2013.

[JBR98] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*, volume 1. Addison-Wesley, 1998.

[JJS91] David Johnson, Roger Johnson, and Karl Smith. *Active learning: Cooperation in the college classroom*. Interaction Book Company, 1991.

[Joh12] Hillary Johnson. How to play the Team Estimation Game, May 2012. Retrieved January 08, 2016 from http://www.agilelearninglabs.com/2012/05/how-to-play-the-team-estimation-game.

[JT79] Randall Jensen and Charles Tonies. *Software Engineering*. Prentice Hall, 1979.

[KA14] Stephan Krusche and Lukas Alperowitz. Introduction of Continuous Delivery in Multi-Customer Project Courses. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 335–343. IEEE, 2014.

[KABW14] Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin Wagner. Rugby: An agile process model based on continuous delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 42–50. ACM, 2014.

[Kay90] Thomas Kayser. *Mining group gold: How to cash in on the collaborative brain power of a group*. Serif Publishing, 1st edition, 1990.

[KB14] Stephan Krusche and Bernd Bruegge. User feedback in mobile development. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, pages 25–26. ACM, 2014.

[KBB16] Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. Teaching Code Review Management using Branch Based Workflows. In *Companion Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016.

[KDoD+03] Lena Karlsson, Å Dahlstedt, J Natt och Dag, Björn Regnell, and Anne Persson. Challenges in market-driven requirements engineering-an industrial interview study. In *Proceedings of the Eighth International Workshop on Requirements Engineering: Foundation for Software Quality*, pages 101–112, 2003.

[Kel91] Marc Kellner. Software process modeling support for management planning and control. In *Proceedings of the 1st International Conference on the Software Process*, pages 8–28. IEEE, 1991.

[KK05] Eeva Kangas and Timo Kinnunen. Applying user-centered design to mobile application development. *Communications of the ACM*, 48(7):55–59, 2005.

[KKB16] Sebastian Klepper, Stephan Krusche, and Bernd Bruegge. Semi-automatic generation of audience-specific release notes. In *Proceedings of the 1st International Workshop on Continuous Software Evolution and Delivery*. IEEE/ACM, 2016.

[KKLK05] Sari Kujala, Marjo Kauppinen, Laura Lehtola, and Tero Kojo. The role of user involvement in requirements quality and project success. *13th International Conference on Requirements Engineering*, 2005.

[KKLW01] Jos Korthagen, Fredand Kessels, Bob Koster, Bram Lagerwerf, and Theo Wubbels. *Linking practice and theory: The pedagogy of realistic teacher education*. Routledge, 2001.

[KKP+15] Sebastian Klepper, Stephan Krusche, Sebastian Peters, Bernd Bruegge, and Lukas Alperowitz. Introducing continuous delivery of mobile apps in a corporate environment: A case study. In *Proceedings of the 2nd International Workshop on Rapid Continuous Software Engineering*, pages 5–11. IEEE/ACM, 2015.

[Kol84] David Kolb. *Experiential learning: Experience as the source of learning and development*, volume 1. Prentice Hall, 1984.

[KRTB16] Stephan Krusche, Barbara Reichart, Paul Tolstoi, and Bernd Bruegge. Experiences from an experiential learning course on games development. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 582–587. ACM, 2016.

[Kru04] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley, 2004.

[Kuj03] Sari Kujala. User involvement: a review of the benefits and challenges. *Behaviour & information technology*, 22(1):1–16, 2003.

[Kö06] Peter Köhler. *ITIL: Das IT-Servicemanagement Framework*. Springer Science & Business Media, 2006.

[LB85] Meir Lehman and Laszlo Belady. *Program evolution: processes of software change*. Academic Press, 1985.

[LKLB16] Yang Li, Stephan Krusche, Christian Lescher, and Bernd Bruegge. Teaching global software engineering by simulating a global project in the classroom. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 187–192. ACM, 2016.

[LMP+15] Marko Leppanen, Simo Makinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika Mantyla, and Tomi Mannisto. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015.

[Mah90] Michael Mahoney. The roots of software engineering. *CWI Quarterly*, 3(4):325–334, 1990.

[Mar07] Matthias Marschall. Transforming a six month release cycle to continuous flow. In *Proceedings of the Agile Conference*, pages 395–400. IEEE, 2007.

[MBDP+14] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495. ACM, 2014.

[MBNC14] Kıvanç Muşlu, Christian Bird, Nachiappan Nagappan, and Jacek Czerwonka. Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes. In *Proceedings of the 36th International Conference on Software Engineering*, pages 334–344, 2014.

[MDH13] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of CSCW*, pages 117–128. ACM, 2013.

[Men08] Tom Mens. *Introduction and roadmap: History and challenges of software evolution*. Springer, 2008.

[MFS15] Martin Michlmayr, Brian Fitzgerald, and Klaas-Jan Stol. Why and how should open source projects adopt time-based releases? *Software, IEEE*, 32(2):55–63, 2015.

[MHR09] Walid Maalej, Hans-Jörg Happel, and Asarnusch Rashid. When users become collaborators: towards continuous and context-aware user input. In *Proceedings of the 24th SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 981–990. ACM, 2009.

[Mor04] Mario Moreira. *Software configuration management implementation roadmap*, volume 1. John Wiley & Sons, 2004.

[Mou05] Mountain Goat Software. Scrum images, 2005. Retrieved January 08, 2016 from https://www.mountaingoatsoftware.com/agile/scrum/images.

[MP11] Walid Maalej and Dennis Pagano. On the Socialness of Software. In *9th International Conference on Dependable, Autonomic and Secure Computing*, pages 864–871. IEEE, 2011.

[ND86] D. Norman and S. Draper. *User centered system design; new perspectives on human-computer interaction*. Erlbaum, 1986.

[NH93] J. Nielsen and J. Hackos. *Usability engineering*. Academic Press, 1993.

[NR69] Peter Naur and Brian Randell. Software Engineering: Report of the 1968 conference in Garmisch, Germany. *NATO Software Engineering Conference*, 1969.

[NRB76] Peter Naur, Brian Randell, and John Buxton. *Software engineering: concepts and techniques: proceedings of the NATO conferences*. Petrocelli/Charter, 1976.

[NS13] Steve Neely and Steve Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference*, pages 121–128. IEEE, 2013.

[OAB12] Helena Olsson, Hiva Alahyari, and Jan Bosch. Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *38th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 392–399. IEEE, 2012.

[Obj08a] Object Management Group. *Business Process Definition Meta Model Specification (Version 1.0)*, 2008.

[Obj08b] Object Management Group. *Software & Systems Process Engineering Meta Model Specification (Version 2.0)*, 2008.

[OR94] Babatunde Ayodeji Ogunnaike and Willis Harmon Ray. *Process Dynamics, Modeling, and Control*, volume 1. Oxford University Press, 1994.

[Pag11] Dennis Pagano. Towards systematic analysis of continuous user input. In *Proceedings of the 4th International Workshop on Social Software Engineering*, pages 6–10. ACM, 2011.

[Pag13] Dennis Pagano. *PORTNEUF-A Framework for Continuous User Involvement*. PhD thesis, Technische Universität München, 2013.

[Pat05] Ron Patton. *Software Testing*. Sams, 2005.

[PB13] Dennis Pagano and Bernd Bruegge. User involvement in software evolution practice: a case study. In *Proceedings of the 35th International Conference on Software Engineering*, pages 953–962. IEEE, 2013.

[PCL+04] Stanley Presser, Mick Couper, Judith Lessler, Elizabeth Martin, Jean Martin, Jennifer Rothgeb, and Eleanor Singer. Methods for testing and evaluating survey questions. *Public opinion quarterly*, 68(1):109–130, 2004.

[PM13] Dennis Pagano and Wiem Maalej. User feedback in the appstore: An empirical study. In *Proceedings of the 21st IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, 2013.

[PP06] Mary Poppendiek and Tom Poppendiek. *Implementing Lean Software Development: From Concept to Cash*. Addison Wesley, 2006.

[RB13] Peter Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[RCP+12] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. Contemporary peer review in action: Lessons from open source development. *IEEE*, 29(6):56–61, 2012.

[Rei09] Donald Reinertsen. *The principles of product development flow: second generation lean product development*. Celeritas, 2009.

[RHL+16] Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 2016.

[Roe15] Tobias Roehm. *The MALTASE Framework For Usage-Aware Software Evolution*. PhD thesis, Technische Universität München, 2015.

[Rol93] Colette Rolland. Modeling the requirements engineering process. *Information Modelling and Knowledge Bases*, 1993.

[Roy70] Winston Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, 1970.

[RS11] David Rosenwasser and Jill Stephen. *Writing analytically*. Cengage Learning, 2011.

[RSI96] J. Rudd, K. Stern, and S. Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, 1996.

[SB02] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Prentice Hall, 2002.

[SB14] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

[Sca01] Walt Scacchi. Process models in software engineering. *Encyclopedia of software engineering*, 2001.

[Sch95] Ken Schwaber. Scrum development process. In *Proceedings of the OOPSLA Workshop on Business Object Design and Information*, 1995.

[Sch11] Kurt Schneider. Focusing Spontaneous Feedback to Support System Evolution. In *19th International Requirements Engineering Conference*, pages 165–174. IEEE, 2011.

[Sco02] Kendall Scott. *The unified process explained*. Addison-Wesley, 2002.

[Sha96] Mary Shaw. Some patterns for software architectures. *Pattern languages of program design*, 2:255–269, 1996.

[SKCU77] Y Sugimori, K Kusunoki, F Cho, and S Uchikawa. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *The International Journal of Production Research*, 15(6):553–564, 1977.

[SLS14] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, 2014.

[SMP+10] Kurt Schneider, Sebastian Meyer, Maximilian Peters, Felix Schliephacke, Jonas Mörschbach, and Lukas Aguirre. Feedback in Context: Supporting the Eolution of IT-Ecosystems. In *Product-Focused Software Process Improvement*, pages 191–205. Springer, 2010.

[Ste10] Angus Stevenson. *Oxford Dictionary of English*. Oxford University Press, 2010.

[STG03] Reinhard Sefelin, Manfred Tscheligi, and Verena Giller. Paper prototyping-what is it good for?: a comparison of paper-and computer-based low-fidelity prototyping. In *CHI'03 extended abstracts on Human factors in computing systems*, pages 778–779. ACM, 2003.

[Suc07] Lucy Suchman. *Human-machine reconfigurations: Plans and situated actions*. Cambridge University Press, 2007.

[Tay14] Frederick Taylor. *The principles of scientific management*. Harper, 1914.

[TDH14] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th international conference on Software engineering*, pages 356–366. ACM, 2014.

[TF99] Shawn Tseng and BJ Fogg. Credibility and Computing Technology. *Communications of the ACM*, 42(5):39–44, 1999.

[TN86] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard business review*, 64(1):137–146, 1986.

[Ver15] VersionOne. 9th annual state of agile development survey, 2015. Retrieved January 08, 2016 from https://www.versionone.com/pdf/state-of-agile-development-survey-ninth.pdf.

[WF84] Gerald Weinberg and Daniel Freedman. Reviews, walkthroughs, and inspections. *IEEE Transactions on Software Engineering*, SE-10(1):68–72, 1984.

[WK02] Laurie Williams and Robert Kessler. *Pair programming illuminated*. Addison-Wesley, 2002.

[Won84] Carolyn Wong. A successful software development. *IEEE Transactions on Software Engineering*, pages 714–727, 1984.

[Wri12] Hyrum Wright. *Release engineering processes, their faults and failures*. PhD thesis, University of Texas, 2012.

[WS02] C. Walrad and D. Strom. The Importance of Branching Models in SCM. *Computing Practices*, 2002.

[Wuj10] Tom Wujec. The Marshmallow Challenge - TED Talk, 2010. Retrieved January 08, 2016 from http://marshmallowchallenge.com.

[XKB15] Han Xu, Stephan Krusche, and Bernd Bruegge. Using software theater for the demonstration of innovative ubiquitous applications. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 894–897. ACM, 2015.

[YF07] Yunwen Ye and Gerhard Fischer. Designing for participation in socio-technical software systems. In *Universal Acess in Human Computer Interaction. Coping with Diversity*, pages 312–321. Springer, 2007.

[You79] Edward Yourdon. *Structured walkthroughs*. Prentice Hall, 1979.

[ZPB+10] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.