Managing Academic Eclipse-Based Projects

Diploma Thesis

Tim Grebien

August 17th, 2012

Christian-Albrechts-Universität zu Kiel
Department of Computer Science
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

Advised by: Christoph Daniel Schulze

Abstract

Sustaining and improving software quality has always been a challenge for large software projects. This applies even more to academic software projects with their experimental nature, fast-changing requirements and changing developers. In this thesis we first learn about well-defined processes for software development that are used for commercial software projects, and then discover why these processes cannot be adapted easily to academic projects.

Then we define a software process for KIELER, an academic research software project developed by the Real-Time and Embedded Systems Group at Kiel University. The approach that will be taken is to define processes for various aspects of software project management, such as issue tracking, build management and software releases. Together this will form the new KIELER software process.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Contents

List of Figures

Acronyms and Abbreviations

API	Application Programming Interface
CBI	Common Build Infrastructure
CI	Continuous Integration
GTK	GIMP Toolkit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IM	Instant Messaging
JAR	Java Archive
JDK	Java Development Kit
JSP	Java Server Pages
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
OSGi	Open Services Gateway initiative
PDE	Plug-In Development Environment
POM	Project Object Model
RCA	Rich Client Application
RCP	Rich Client Platform
SCM	Source Code Management
SVN	Subversion
SWT	Standard Widget Toolkit
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

Chapter 1

Introduction

Managing software projects has been a field of research since the mid 1960s. While in the early years of computer engineering there was a clear focus on building hardware, the term of *software engineering* first appeared in the late 1950s and early 1960s. With fast growing hardware complexity, software development could not keep the pace, causing many projects to exceed their budget and also their schedule. In 1968 the NATO Science Committee sponsored the first two conferences that explicitly focused on software engineering [NR69]. At this time people realised that a certain focus has to be set on the development process when designing and implementing software. However this *software crisis*, as Dijkstra called it, was not to be solved soon [Dij72]. In the following years new approaches were developed. There was the introduction of new programming paradigms, such as object-oriented programming or structured programming. Others advocated the use of defined processes and development models, such as the *Waterfall Model* or the *Capability Maturity Model*. With the introduction of formal methods some people believed that by applying formal engineering methodologies, software development would become predictable by proving programs correct.

Nowadays, it is consensus that there is no silver bullet in software engineering [BJ87]. The field is too complex and too diverse to solve all problems in software engineering with just one method. In present times, as computers are omnipresent, the demand for smaller software solutions has raised. Therefore, to keep software development as inexpensive as possible, the need for faster and simpler methodologies has led to lightweight software processes. Modern trends in software engineering include *Agile Software Development*, *Model Driven Design* and *Software Product Lines*.

Outline

This thesis will give insight on how software engineering can be applied to large academic software projects. We will learn about the differences between commercial software development and software development in the academic environment. In chapter 2 technologies that will be used will be presented. This includes the presentation of the Eclipse platform, the *Kiel Integrated Environment for Layout Rich Client Platform* (KIELER), and technologies that support the development process. KIELER is an Eclipse Rich Client Application (RCA) developed by the *Real-Time and Embedded Systems Group* at Christan-Albrechts-University Kiel. The development process this thesis introduces will be applied to KIELER. In chapter 3 software engineering methodologies and development processes that are commonly used in commercial software development are presented. We will learn about the problems that arise when trying to apply such processes to academic software projects. Afterwards, in chapter 4 a developers perspective on KIELER is given. As part of the work for this thesis, software developers that work on KIELER were interviewed to get an insight about how developers think about KIELER development. This is compared to the thoughts of both developers in other academic projects and in commercial projects which have also been interviewed.

The third part, starting with chapter 5, then defines processes for various subareas in the development of KIELER. A headless build process is implemented, issue tracking is described, a code review process for KIELER is introduced, and guidelines on how sources are kept and how releases are conducted are given. Afterwards, in chapter 10 and 12, all of this is summarised to a process that explains how future software development in KIELER is done. Finally, in chapter 11 and 12 these results are evaluated and ideas are given on how KIELER development can be improved even further in the future.

1.1 Related work

Over the years, much research has been done on the process of developing software. Beginning with the ideas of Royce [Roy70] on the waterfall model and later Boehm [Boe88] on the spiral model up to today's popular agile, lightweight processes, software engineering has involved over time. However, in this thesis we will learn why such processes are not applicable to academic software processes.

Regarding methods for academic software development in particular only little research has been done. This is likely due to the fact that academic software projects are mainly demonstrators for the research work that is done in PhD or student theses. Often the lifecycle of academic software projects is only over the course of one PhD thesis [JBH+10]. In 1999 Reekie et al. described their efforts at improving software development in the Ptolemy[1] project [RNHL99]. Ptolemy is an academic software project at UC Berkeley. In their paper they describe in detail how software development in the Ptolemy project is done. Their work is the primary source of inspiration for this thesis. However, Reekie describes a rather formal and heavyweight approach which would be too work-intensive to introduce for the KIELER project. Also, during the last thirteen years, the tool support for software development has advanced. This thesis will propose a more lightweight approach that is supported by modern project management tools, with some inspiration taken from the Ptolemy process.

This thesis will introduce a lightweight management process for the academic software project KIELER. Therefore research in agile development methods is also relevant. The SCRUM process which is described by Schwaber and Beedle explains how to develop software in a flexible process that works well with fast-changing requirements [SB01]. While SCRUM is not applicable to KIELER as whole, the idea of having a project secretary is inspired by the SCRUM process.

Other related work deals mainly with special topics that will be described in this thesis. They are mainly used as a reference to support and emphasise the importance of the different parts of a software process this thesis will introduce.

Bertram et al. gave insight on how issue tracking can improve communication in development teams [BVGW10] while Bettenburg et al. discuss the quality of bug reports and their improvement [BJS+08].

Regarding continuous integration, Fowler describes principles of improving the software stability by continuously building and testing the source code after each repository change [Fow06]. Together with the ideas of Duvall et al. [DMG07] it forms the basis for the KIELER continuous integration process. However, there is no integration environment for KIELER and since development in the KIELER project relies heavily on branches their described process has to be adapted accordingly.

[1] http://ptolemy.eecs.berkeley.edu

In the field of source code reviews Rigby and German described how they are done in various open source projects [RG06]. While the code review process of large and distributedly developed open source software projects cannot be applied directly to the KIELER project, it gave inspiration and also confidence that conducting source code reviews is a proper measure to improve source code quality.

Chapter 2

Technologies

This chapter presents tools and frameworks that are either used to implement the solutions proposed in this thesis or technologies that KIELER is built on. KIELER is an Eclipse Rich Client application; therefore the Eclipse platform will be discussed first and afterwards KIELER is introduced.

2.1 The Eclipse Platform

When reading about Eclipse[1] it is often described as an *integrated development environment* (IDE) for developing Java applications. However, this is only part of the truth. Starting as a project at IBM Canada as a successor to the VisualAge product family, it has been released to the open source community in 2004 with the establishment of the Eclipse Foundation. Eclipse is a component-based development framework that is extensible through plugins. By the use of plugins it can be used to implement software in nearly all popular programming languages including Java, C/C++, Haskell, Perl, etc.

From a technical point of view, Eclipse is a service platform that allows development of arbitrary applications based on the extensible Eclipse component model [IBM06]. Since version 3.0 the core of Eclipse is *Equinox*, a certified implementation of the *Open Services Gateway initiative* (OSGi) framework. OSGi is a modular service platform for Java that implements a dynamic component model [OSG07]. Figure 2.1 shows the OSGi architecture with its distinct

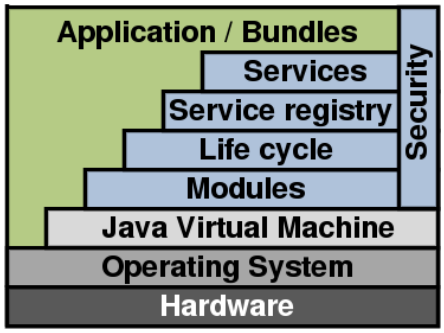


Figure 2.1. OSGi Architecture (Image by F. Akeel, public domain)

layers. Equinox implements the following layers: *Services*, *Service registry*, *Life cycle*, *Modules* and *Security*.

Implementation of components in the OSGi model is done in bundles. On the Eclipse Platform such OSGi bundles are called plugins. An Eclipse plugin is a group of Java classes and

[1] http://www.eclipse.org

other resources, such as images, that also contains metadata in `MANIFEST.MF` and `plugin.xml` files.

MANIFEST.MF

Manifest files contain OSGi metadata that is stored in key-value pairs. An example manifest file taken from the KIELER sources looks like this:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: %pluginName
Bundle-SymbolicName: de.cau.cs.kieler.core.kgraph;singleton:=true
Bundle-Version: 0.4.1.qualifier
Bundle-ClassPath: .
Bundle-Vendor: %providerName
Bundle-Localization: plugin
Export-Package: de.cau.cs.kieler.core.kgraph,
  de.cau.cs.kieler.core.kgraph.impl,
  de.cau.cs.kieler.core.kgraph.util
Require-Bundle: de.cau.cs.kieler.core,
  org.eclipse.emf.ecore;bundle-version="2.5.0";visibility:=reexport
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

As it can bee seen in the example, the metadata a manifest file contains mostly package-specific information such as `Bundle-Name` or `Bundle-Version`. It can also be observed that dependency information is stored in the manifest file as well.

plugin.xml

The `plugin.xml` file stores Eclipse-specific metadata that extends the OSGi specification. Eclipse plugins can for example define *Extension Points* that other plugins can plug into by defining *Extensions*. Extensions are defined in `plugin.xml` as the following example shows:

```
<extension point="org.eclipse.emf.ecore.generated_package">
   <package
         uri="http://kieler.cs.cau.de/KGraph"
         class="de.cau.cs.kieler.core.kgraph.KGraphPackage"
         genModel="model/kgraph.genmodel"/>
</extension>
```

An extension is defined for an extension point in the plugin `org.eclipse.emf.ecore.generated_package`. Dependencies can also be defined in `plugin.xml`, however this has rather historical reasons: before the release of version 3.0 Eclipse was based on a proprietary component framework which allowed the specification of dependencies only through `plugin.xml`.

Other Eclipse packaging types

In addition to plugins, the Eclipse platform also allows the creation of features. Features are comprised of a set of references to Eclipse plugins. These references are defined in the file `feature.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="de.cau.cs.kieler.core.annotations.feature"
      label="KIELER Annotations"
      version="0.4.0.qualifier"
      provider-name="Christian-Albrechts-Universitaet zu Kiel"
      plugin="de.cau.cs.kieler.core.annotations">

   <url>
      <update label="KIELER Nightly Builds"
            url="http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/nightly/"/>
   </url>
   <requires>
      <import feature="com.google.guava.runtime.feature" version="10.0.0"
                     match="greaterOrEqual"/>
      <import feature="org.eclipse.emf" version="2.7.0" match="greaterOrEqual"/>
      <import feature="org.eclipse.xtext.runtime" version="2.1.0"
                     match="greaterOrEqual"/>
      <import feature="org.eclipse.xtext.ui" version="2.1.0" match="greaterOrEqual"/>
   </requires>

   <plugin
         id="de.cau.cs.kieler.core.annotations"
         download-size="0"
         install-size="0"
         version="0.0.0"
         unpack="false"/>

   <plugin
         id="de.cau.cs.kieler.core.annotations.edit"
         download-size="0"
         install-size="0"
         version="0.0.0"
         unpack="false"/>
</feature>
```

The example above is an excerpt from the KIELER annotations's `feature.xml`. The file contains a `<requires>` section that indicates on which other features this particular feature depends. Furthermore the `<plugins>` section defines which plugins the feature is comprised of.

Features are typically installed from Eclipse P2 repositories. An Eclipse P2 repository hosts a set of Eclipse plugins and features, usually reachable via HTTP. Besides plugins and features, a P2 repository contains two XML files, `artifacts.xml` and `contents.xml`, that describe the layout, and contents of the P2 repository. When a user wants to extend Eclipse by installing new features, he may enter the URL of a P2 repository in the Eclipse *Install new Software* dialogue and is presented with all features the update site contains. As dependencies are recorded in `feature.xml`, automatic dependency resolution is possible.

2.1.1 The Eclipse Plugin Development Environment

For the implementation of Eclipse plugins, features, and rich client applications, the *Eclipse Plugin Development Environment* (PDE) provides tools to assist the developer in the creation, development, test, debug, and build process. The PDE is comprised of three main components: *PDE UI*, *PDE build*, and the *PDE API Tools*.

PDE UI

PDE UI provides the user interface tools in the Eclipse IDE to help the user develop Eclipse plugins and OSGi bundles. There are editors for metadata files, such as `MANIFEST.MF` and `plugin.xml`. When converting plain Java projects to Eclipse plugins PDE UI provides conversion tools to assist the developer with the import. Furthermore, there are multiple other wizards for project creation, plugin export, or for the generation of Eclipse Rich Client Applications

PDE build

PDE Build is used for the automation of plugin build processes. To facilitate this, PDE build generates scripts that can be processed by the build automation tool Ant. PDE build can also generate scripts to build Eclipse features, Eclipse applications, and P2 repositories.

PDE API Tools

The PDE API Tools assist developers in providing a stable API for their plugins. The PDE API Tools notify the developer if binary incompatibilities between two versions of the same software component occur. Furthermore, they assist the developer in increasing the plugin versions if changes to the API have been made. This is done by doing an API scan that defines the baseline API first. If the API changes from the baseline the developer is informed that the version has to be increased.

2.1.2 The Eclipse Rich Client Platform

The Eclipse Rich Client Platform (RCP) enables developers to implement arbitrary applications on top of the Eclipse runtime. Basically, the RCP is the minimum set of components needed to run Eclipse. The RCP is constituted of the following components:

- Equinox
- The Core Platform
- The Standard Widget Toolkit (SWT)
- JFace

Equinox together with the Core Platform forms the Eclipse runtime. In pre-OSGi days, before the release of Eclipse 3.0, the Eclipse Core Platform was the foundation of the Eclipse IDE. Nowadays, much functionality has moved to the OSGi implementation Equinox. Today, the Core platform's primary function is to boot the OSGi runtime and also to provide services that extend the OSGi framework, such as plugin extension points.

The Standard Widget Toolkit (SWT) is a library for the implementation of graphical user interfaces in Java and is maintained by the Eclipse Foundation. Acting as a wrapper around

native window toolkits such as GTK on Linux or Cocoa on Mac OS, SWT allows the developer to design user interfaces for platform-independent applications that provide the native look and feel of the operating system they are run on. JFace is built on top of SWT to help the developer in implementing complex user interfaces by combining the SWT base components. JFace implements the model view controller pattern that basically separates the representation of information from interaction.

Applications developed on top of the Eclipse RCP are called *Rich Client Applications* (RCA). Eclipse RCAs can also use arbitrary Eclipse components, such as Wizards or UI components, by loading the respective plugins.

2.1.3 The Eclipse Common Build Infrastructure

The Eclipse Common Build Infrastructure (CBI) is an initiative by the Eclipse Foundation to describe a common build process for Eclipse plugins and applications. The CBI is primarily aimed at Eclipse-based projects that are managed by the Eclipse foundation. Until now, most projects on `eclipse.org` implement their own build process. This means that developers that want to contribute to different Eclipse projects have to learn about build-specific details not relevant to their actual contribution before they can start work. The CBI suggests a common build process centred around *Hudson*, *Maven*, and *Tycho* which will all be described in detail later on.

2.2 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client Platform (KIELER)[2] is a research project developed at the Real-Time and Embedded Systems Group at the Christian-Albrechts-University in Kiel. KIELER focuses on enhancing the graphical model-based design of complex systems. The project was founded in 2008 to implement the ideas on pragmatics of model-based design by Fuhrmann and von Hanxleden [FvH08]. KIELER is available as open source software licensed under the EPL license. The main focus of KIELER is to research and develop new concepts and methods for designing and editing different types of diagrams, such as SyncCharts, UML diagrams, or data flow diagrams. KIELER aims at minimising time and effort to generate such diagrams by optimising the development interface.

A concept introduced through KIELER is for example *structure-based editing*. Classical modelling tools facilitate the usage of drag and drop for editing the graphical models. While this is well-known to developers, studies have shown that developers have to perform certain *enabling steps*, for example freeing space before adding a new component, when changing an existing model. The idea of structure-based editing is to provide a set of pre-defined transformations to perform editing tasks, such as adding a component to the diagram. When performing such a transformation, automatic layout has to be applied to the diagram afterwards to free the developer from the aforementioned enabling steps. Therefore, an important field of research in KIELER is also the development of layout algorithms.

2.2.1 KIELER from a Technical Point of View

As this thesis focuses on improving the software development process of KIELER, we have to take a look at the technical aspects of its development. Starting from 2008, 61 developers have

[2] `http://www.informatik.uni-kiel.de/kieler`

committed over three million lines of source code in 13407 commits. Even if the line count also includes metadata and generated source code, KIELER can be considered a large project. According to Ohloh[3], a website that rates open source software projects, KIELER is developed by one of the largest open source teams in the world.

Based on the Eclipse Rich Client Platform, KIELER is comprised of 171 Eclipse plugins that are made available in 19 features. Compared to other actively developed large open source projects, KIELER can be safely considered a huge Eclipse-based project. For comparison, the Apache Directory Studio, a management application for LDAP servers, is comprised of only 30 plugins. Looking at the large amount of plugins and developers, it becomes clear that good project management and a well-defined software process is needed to keep KIELER's growing complexity under control.

When looking into the details of KIELER, it becomes clear how the project has grown that large. Effectively, the KIELER project can be divided into four parts as in Figure 2.2.

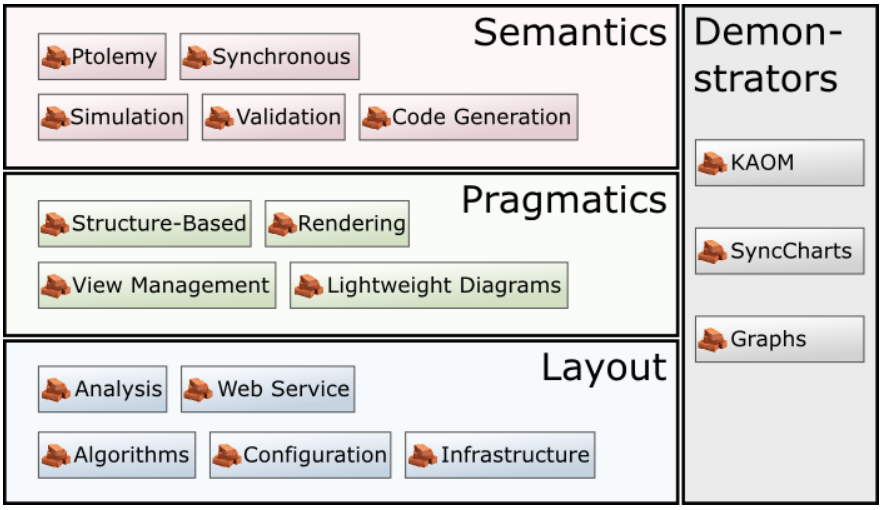


Figure 2.2. KIELER project structure (Image from the KIELER documentation)

The *Layout* part contains the KIELER graph and diagram layout infrastructure. The *Pragmatics* part focuses on development of new ideas in the graphical representation and editing of Diagrams, while the *Semantics* part provides simulators and code generation for graphical modelling languages.

For the presentation of the underlying parts, the KIELER *Demonstrators* part contains graphical editors and simulation environments.

2.3 Other Technologies

Apart from Eclipse and KIELER some other technologies that will support the software development process have to be mentioned. This section will give a brief introduction to all software that is used to support the KIELER development process. Some software and technologies will also be described in more detail later.

[3] `http://www.ohloh.net/p/kieler/factoids`

Extensible Markup Language

The Extensible Markup Language (XML) is a markup language is designed to encode documents in a human-readable and machine-readable format. As the name suggests XML is designed to be extensible in any form. While XML provides a small set of syntax rules, the contents of an XML document are described through a document type definition. This extensibility, together with both human- and machine-readability, makes XML well suited for software configuration files.

Apache Tomcat

Apache Tomcat is an open source web server and also a servlet container designed to run server-side Java code. Tomcat implements Oracle's Java Servlet and Java Server Pages (JSP) specifications. This enables development of Java backends for web interfaces to generate dynamic web applications that run on the Tomcat server. Strictly speaking Tomcat is comprised of three parts, *Catalina* the servlet container, *Coyote* the HTTP server, and *Jasper* the JSP engine.

Git

Git is a distributed *source code management* (SCM) system. It has been originally developed to manage the sources of the Linux kernel. Git is designed for strong branching and merging support and also with high performance in mind.

Maven

Maven is an open source build and project management tool developed by the Apache Foundation. Maven is designed to build Java applications and can also be used to generate project websites and documentation pages through Javadoc. Maven utilises a component-based approach, all dependencies and even the Maven core plugins are fetched from remote or local Maven repositories. Maven introduces a *Project Object Model* (POM) for compiling and managing Java projects, which is defined in XML files.

Tycho

Tycho is a set of Maven plugins designed to compile Eclipse plugins, features, and rich client applications. This is achieved by making Eclipse metadata available to Maven. Originally developed by Sonatype, Tycho has been released as open source and is now developed as an Eclipse incubator project by the Eclipse Foundation.

Ant

Ant is an extensible open-source Java build tool for automating build processes, developed by the Apache Foundation. The Ant build process is described in XML files in which build targets that are made up from multiple tasks are defined.

Continuous Integration

Continuous integration (CI) describes a process of frequently applying code changes to the authoritative source code repository. After source code checkins the codebase is built and

tested automatically by a CI server. Typical CI conventions expect developers to commit their code changes at least daily and not to break the build process.

Issue Tracking

In software development it is unavoidable that the developed software sometimes shows unwanted behaviour. To keep track of such bugs and also of other open development tasks in collaborative software development, issue tracking systems are used. Issue tracking systems allow to store such incidents in tickets that contain details about a single bug or a single task.

Unit Tests

Unit testing is a method to test small isolated pieces of source code. In Java programming a unit that is tested is typically a class or even a method. In Java the test framework *JUnit* is often used. JUnit allows the developer to write special test classes that implement test cases for the classes that are to be tested and afterwards compare the test output to the expectations implemented in the test class.

Static code analysis

Static code analysis is a method in software testing that analyses source code or byte code and searches for certain failure patterns. Popular code analysis software is for example *PMD*, *Checkstyle*, and *FindBugs*.

Javadoc

Javadoc is a tool that generates API documentation from special Javadoc comments in Java source code. By default the generated documentation format is HTML. However, it is possible to extend and alter the Javadoc output though the implementation of custom doclets or taglets. Taglets extend the set of special Javadoc comments, while custom doclets can manipulate the Javadoc output.

Chapter 3

Software Processes

As software development has been done for many years, several processes have been described formally to streamline the development process. Over the years methods have evolved from conservative, planning-driven approaches to modern, lightweight, and agile methods. Choosing a software process depends highly on the project and also on the structure of the development team: one project may best be developed in a heavyweight process while another project needs a more flexible approach. In this chapter we will learn about some of these processes and we will discuss their application to KIELER development.

3.1 The Waterfall Model

The waterfall model is one of the first software development models that has been described. While it was first mentioned in the 1950s, it was formally described by Royce in 1970 [Roy70]. The waterfall model splits software development into multiple phases; each phase should be fully completed before entering the next phase.

1. Requirements specification
2. Design
3. Implementation
4. Verification
5. Maintenance

Requirements Specification

In the first step of the waterfall model a complete description of system behaviour including use cases that describe user interactions with the system is written. This includes describing non-functional requirements, such as performance requirements or quality standards.

Design

After all requirements are specified, the implementation is planned by specifying a software design. Software design concerns the decomposition of the system into smaller parts. Software design has been described as a wicked problem by DeGrace and Stahl in 1990 [DS90]. There is not just one design solution for a particular problem. In object oriented programming, software design means decomposing the system into a set of classes with associated methods, by applying the fundamental paradigms of object oriented programming: abstraction, encapsulation, inheritance, and polymorphism. The object oriented design is then often specified in the form of one or multiple class diagrams in the UML language.

Implementation

In the implementation phase, the software system is implemented in a programming language. The choice of programming language is mostly a result of the specification and of the design phase. Different programming languages have their own strengths and weaknesses, and therefore differ in their suitability for different types of programming problems.

Verification

In the verification phase the implemented software undergoes a verifying process. First it is verified that all requirements set in the first phase have been implemented. Also, the implementation is validated against the design. Furthermore, the implementation is tested for unwanted behaviour to minimise the number of issues that arise in the maintenance phase.

Maintenance

The maintenance phase concludes software development in the waterfall model. After Lientz and Swanson four types of maintenance can be distinguished [LS80]:

- Corrective maintenance deals with the fixing of software faults.
- Adaptive maintenance is needed to adapt the software to environmental changes, such as new hardware.
- Perfective maintenance is about adapting the system to new requirements, such as functional enhancements or performance improvements.
- Preventive maintenance is about increasing the system's maintainability by for example writing further documentation, cleaning up source code, etc.

3.1.1 Application to KIELER

It is evident that the waterfall model is a planning-driven approach. Each consecutive phase has to be completed before starting the next phase. This leads to problems when considering the application of the waterfall model to KIELER. Being a research project, requirements are subject to change at any time, making it impossible to write a requirements specification for the whole project. Therefore it is highly possible that the maintenance phase will never be reached. Even when considering the application of the waterfall model only to single components of KIELER it will be found to be difficult to impossible. Also the waterfall process is heavily document-driven. Research, however, is more flexible: new methods are considered, tried out, and constantly refined. Therefore, the KIELER project needs a more iterative, less planning-driven process that is more flexible and better adapted to research needs.

3.2 The Spiral Model

Criticism on the waterfall model regarding its strict sequential phases led to the introduction of iterative principles into the software development process. In 1986, Boehm described the popular spiral model [Boe88]. The spiral model combines the idea of iterative development with the aspects of the waterfall model. Iterative in this case means that starting from a prototype there are incremental releases, each introducing new functionality. The spiral model

is designed for large projects with iterations that take between six months to three years. An advantage of the spiral model is the introduction of risk management for the software development cycle. Risk management in each iteration of the spiral helps to avoid project failure in later project stages, in an attempt to identify critical problems as early as possible. Figure 3.1 shows how software development in the spiral model is done. Basically each iteration of the cycle is comprised of four steps, one for each quadrant.

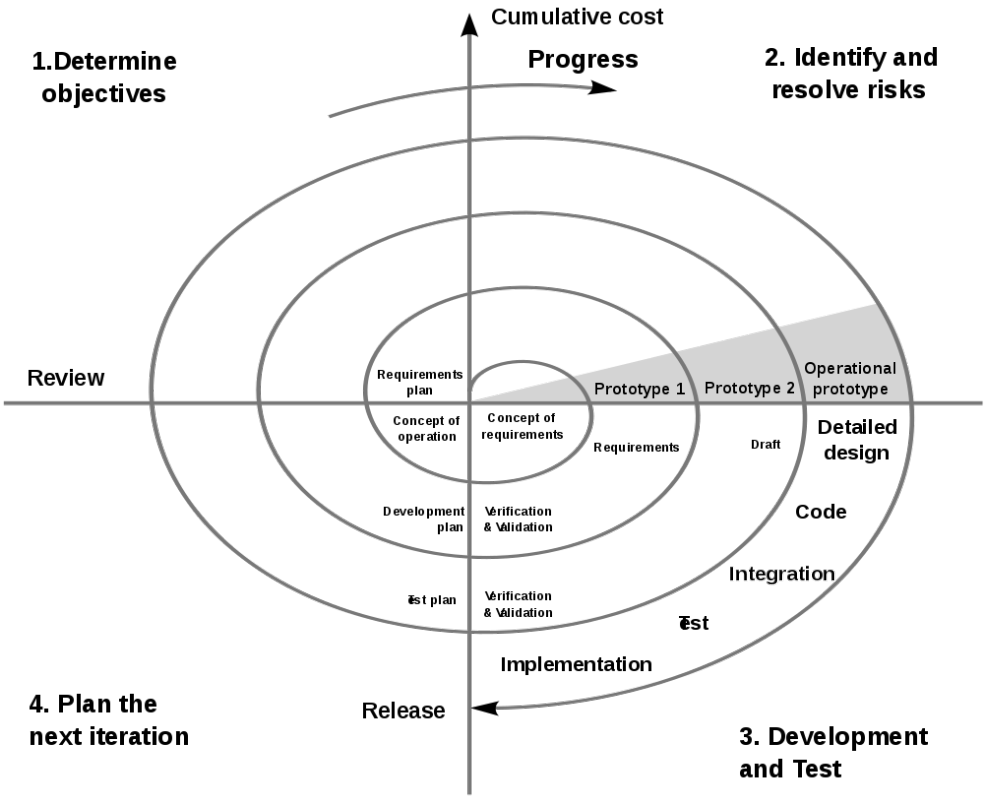


Figure 3.1. The spiral model (Image public domain)

Determine Objectives

In the first phase objectives for the current iteration are set. This includes the identification of alternatives and also the description of constraints. This can be compared to the requirements engineering phase of the waterfall model, with the difference that no full requirements specification is needed as by revolution of the spiral each phase is conducted multiple times.

Identify and Resolve Risks

In this phase the alternatives that have been found in the first phase are evaluated. Risks are identified and rated. In the spiral model a risk is rated by the possibility that a certain objective or requirement cannot be fulfilled during the implementation process. The spiral

model tries to minimise risks. Therefore, the goal of this phase is to identify the most severe risk, and then to try and find the less risky alternative before implementing it in the next phase. If a risk arises that cannot be solved, the project has failed.

Development and Test

In this phase design and implementation takes place. This is normally done by running through the phases of the waterfall model, beginning with a detailed design specification and ending with the verification. Depending on how many iterations of the spiral model have already been made, the result of this phase can vary between producing a prototype and a finished product.

Plan the next iteration

This phase marks the final phase of the iteration. The milestone implemented in the previous phase is released. The development plan is updated and the last iteration is evaluated. Furthermore, test plans are devised and updated and also the next iteration of the spiral model is scheduled and planned. The transition from this phase to the first phase of the next iteration can be rather flowing as updating the development plan can lead to the determination of objectives for the next iteration.

3.2.1 Application to KIELER

This iterative approach to software development seems to be better suited to the KIELER project. However, there are also problems to overcome when trying to apply the spiral model to development in the KIELER project. First of all, not all parts of KIELER are developed at the same time and at the same pace. This means that while no development in the KIELER layout part takes place, the semantics part might be developed very actively. Progress in the development of KIELER depends on many factors; before conference deadlines, for example, the involved developers are often busy with research and are therefore unable to do any development during that time. While this could be circumvented by the introduction of multiple spirals, one for each part of KIELER, there are also other problems. A large part of development in KIELER is done in student theses. As not all students start their thesis at the same time, timing the spiral process would be nearly impossible. However, this is needed as the spiral model does not allow the requirements to change, for example during the implementation phase, which is nevertheless needed when a new student joins the KIELER team to write a thesis. Also, it has to be recognised that the spiral model is a heavyweight process. There are many administrative tasks that have to be done and especially specifications and requirements have to be written and validated. While it is possible to alter the project goals from iteration to iteration, flexibility is limited. Therefore, the spiral model is also unsuitable for the KIELER project.

3.3 Agile Methods

In the first two sections of this chapter we have learnt that both the waterfall model and the spiral model are not flexible enough for software development in the KIELER project, because

they are too heavyweight. Due to the same reasons in the mid-1990s lightweight methods for software development were introduced.

Lightweight methods became popular in 1999 when Beck published the first article about *Extreme Programming* [Bec99]. In 2001 the *Manifesto for Agile Software Development* has been signed by 17 software developers naming these lightweight methods as *Agile Methods* [BBVB+01]. The Agile Manifesto proposed four principles of agile software development:

- **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 development promotes a flexible approach to software development, changes during development are expected. Teamwork and collaboration is promoted during the whole lifecycle of the project.

3.3.1 Introducing SCRUM

SCRUM is a lightweight, agile software development method that was first presented in 1995 by Schwaber [S+95]. Out of the notion that modern software development projects are becoming too complex, SCRUM is based on three fundamental principles that should reduce complexity:

Transparency All advances and problems in the development process are recorded daily and transparently for all project members.

Evaluation New product functionalities are delivered and evaluated regularly.

Adaption Product requirements are updated after each delivery.

SCRUM introduces many new concepts some of which have to be described first before explaining how the process works as whole.

First of all, project members take different roles: *Product Owner*, *Development Team*, and *ScrumMaster*.

The Product Owner

The product owner represents the customer in the SCRUM process. He describes the desired properties of the finished product and is responsible for conveying a clear product vision to the rest of the team. He meets regularly with the customers and acts as an interface between customers and the rest of the SCRUM team. Together with the developers he writes user stories, a special way of recording planned functionalities together with a time estimate for implementation, for the *product backlog*. Also, the product owner assigns a priority for each story.

The Development Team

The development team implements the planned features in the order that the product owner has determined. Furthermore, the development team estimates the amount of work needed for the implementation of each user story. A typical SCRUM development team is made up of three to nine developers. As the development team is responsible for all development work,

including design, implementation, testing, and documentation, the ideal team should be made up of developers with cross-functional skills. In SCRUM the development team is always addressed as whole; for example, no individual team members are to be criticised for mistakes. Good communication between team members is essential: Cockburn emphasises that information should pass around the team by for example overhearing other team members's discussions [Coc05]. Therefore, the development team should share one large office.

The ScrumMaster

The ScrumMaster is responsible for the success of the process. He works together with the development team without being part of it. He acts as a project secretary and ensures that rules, such as coding rules are adhered to. As the development team is self-organising the ScrumMaster is no superior to the developers; his role is often referred to as a *servant-leader*. An important job of the ScrumMaster is to keep the team focused on development by removing any impediments that hinder development.

The Product Backlog

The product backlog in the SCRUM process is the central repository where requirements are kept. Requirements and features are often described in the form of *user stories*. User stories are a method of describing features [Coh04]. They always take the form of: "As a *role*, I would like to have *feature*, so that I have *advantage*". For a text processor, a possible user story could be: "As an author, I would like the software to open the document that I last edited right after startup, because it then saves me time".

The product owner collects and writes user stories that are put in the product backlog. For each user story, the product owner estimates the implementation effort together with the development team. Also, the product owner assigns a priority to each user story.

The product backlog can be compared to a requirements specification in traditional software processes. However, contrary to traditional specifications, it is neither expected, nor wanted that the product backlog is complete before the implementation starts.

For transparency reasons, the product backlog should be visible for all project members. In SCRUM teams this is often realised using a large LCD screen.

3.3.2 The SCRUM Workflow

The SCRUM process begins with a vision of the project owner together with a short description of possible milestones and a time estimation. Afterwards, the product owner fills the product backlog with requirements and assigns priorities to them.

The software is then implemented in an iterative process. Each iteration in SCRUM is called a *sprint*. All sprints are scheduled for the same duration, typically three to six weeks per sprint. Each sprint begins with a *sprint planning meeting*, during which the team creates the *sprint backlog*.

A sprint backlog contains all features and requirements that are to be implemented during the sprint. The contents of the sprint backlog are derived from the product backlog. However, all items in the sprint backlog have to be solvable within one work day. This means that user stories from the product backlog often have to be split into multiple steps in order to fit into the sprint backlog.

Afterwards the sprint starts and the development team begins implementing the features from the sprint backlog as a self-organised team without any disturbance from the outside. This especially means that the sprint backlog does not have to be changed during a sprint.

Each day during the sprint, a short team meeting called *daily scrum* takes place. All developers and the ScrumMaster are obliged to attend the meeting. The product owner should attend the meeting as well, but this is not absolutely required. In the daily scrum, the development team discusses and organises their daily work. If there are any impediments the team tries to solve them during the meeting or, if they fail, they inform the ScrumMaster or product owner. In order to keep meetings short, Schwaber suggests that each team member should only answer the following three questions [SB01]:

- What have I done for the project since the last daily scrum?
- What do I intend to do until the next daily scrum?
- What hinders me from working as effectively as possible?

The ScrumMaster makes sure that the three questions are answered, and keeps the team from drifting into lengthy discussions.

After the sprint is finished the *sprint review* takes place. In the sprint review the product owner is presented with the results of the last sprint. During the review he tests the software in an integration environment. If the product owner is satisfied, he accepts the sprint results. It is important that the product owner only accepts completely finished and working results. If a feature is only nearly finished or faulty it is not accepted. Such results are recorded in the product backlog with a high priority in order to fix them during the next sprint. The sprint review is moderated by the ScrumMaster.

Directly after the sprint review, the *sprint retrospective* is held. During this meeting the developers reflect the last sprint. Most importantly they should discuss collaboration within the team and think about measures to further improve it. The ScrumMaster acts only as a moderator, he records the outcome of the meeting without presenting any solutions. All changes to the development process that have been devised during the retrospective are implemented in the next sprint.

3.3.3 Agile Methods and KIELER

While agile methods provide a lot of flexibility with respect to changing requirements, they are also all built around a team of developers working together. This not only holds true for the SCRUM process, but also for other agile methods, such as *Extreme Programming* or *Crystal*. This team-centric approach, however, makes the application of these methods on the development in the KIELER project difficult. While there is a team of developers working on KIELER, they mostly do not work together but rather as individuals on their own small parts of KIELER. This makes sprint planning impossible, as development does not happen as part of a team effort. While it is possible in SCRUM to split a large project into multiple SCRUM teams, these teams would often consist of only one developer together with the product owner and the ScrumMaster. The benefit, however, would not justify the expense of maintaining multiple product logs, multiple sprint logs, etc. Therefore, agile, team-centric methods of software development are also not applicable on the KIELER project.

3.4 Conclusion

In the previous sections we have learnt about different approaches to the software development process. Also, it has been argued why the presented methods are unsuitable for the KIELER project. While there are many more software development processes that have not been described in this chapter, none of these methods is applicable to the KIELER project due to at least one of the following reasons:

- The method cannot deal with fast-changing requirements.
- The method is too team-centric.
- The method involves a high administrative overhead.

Looking for reasons, it becomes evident that software processes are designed for commercial software development. All methods target the delivery of a finished product to a customer. In academic software development, however, it is often not even expected that the software is ever finished. The software is rather regarded as a testbed or demonstrator for research. Also, there is no customer, and the software does not have to be successful economically.

Furthermore, in commercial software development there is always a team of developers which is employed solely to do development tasks. In the academic environment, software development is only part of the duties of researchers and students. Therefore, proven processes for commercial software development cannot be easily adapted to academic software projects.

For the KIELER project, this means that we will have to find our own process of software development. In the following chapters of this thesis, selected parts of software project management will be presented. Together, they will form the new KIELER software process.

Chapter 4

Developers Perspective on Academic Software

When defining a new development process it is a good idea to learn about how the developers feel about the current process, and even more what they would like to have in the future. This becomes even more important in an academic environment as all developers have a wide range of other responsibilities. If not all developers accept and live the process, it will only be a matter of time until the process itself breaks. Therefore a series of interviews has been conducted asking developers about how they see the current software process. The main objective of those interviews is to find out about things that are running well, and things that should be improved in KIELER. To maintain comparability interviews were done in a semi-structured way: there was a fixed questionnaire of about fifty questions which were asked in a one-to-one dialogue. This left space to emphasise some topics varying from developer to developer. Furthermore, similar interviews with a project-lead from another academic software project and with developers in software companies were carried out. This allows comparison between project management in KIELER and other academic or commercial projects.

4.1 KIELER Developers

The questionnaire for the developers is roughly divided into three sections. In the first section, developers were asked about some general things such as how long they are part of the KIELER team, on which part of KIELER they work, and about their past experience with software development in teams. The second section deals with project organisation. It focuses on how they see the current organisation regarding how well KIELER is documented, how ticketing is used, about the release process, and about how KIELER is built. The third section asked mostly about the code itself. It tries to find out about how developers use the source code management system, how they think about software tests, how they would judge the code quality, and what each developer does to improve his own code's quality.

During the interview sessions, eight developers have been interviewed. All developers have participated in KIELER development for at least half a year, one has been part of the team since the start of the project in 2008, while the average lies at about one and a half years. When asked about former experience in development teams, all developers stated that before working on KIELER they have had at best little experience, mostly in private projects or smaller student projects. No one has had experience with a structured and formal software development process before.

The second part of the interview started with questions concerning communication between team members, which has been generally considered as good. However, nearly all developers stated that it could be improved further. Examples for improvement included

the wish for more communication on the mailing list, especially about interface changes which might break code in other parts of KIELER, and more detail about technical issues in KIELER's various plug ins. One developer was of the opinion that if one looks at the KIELER team on the whole, communication is rather bad. He noted that most developers only work on their small part of KIELER without having an overview of the whole project. This is due to the academic nature of the project where most of the development is done as part of student theses. All developers stated that the weekly KIELER meetings help to improve communication, to get an idea about what is going on in KIELER development, and on which parts other people currently work. Being asked about what could be improved in the meetings, all developers stated that they would like meetings to be more structured. The agenda should be fixed well before the actual meetings are held so that all participants have the chance to prepare properly for the meeting. Also meetings should not lead to lengthy discussions about highly specialised topics concerning only few developers. However, this was stated to have improved over time. The weekly frequency of meetings is seen as fine, only one developer stated that meetings every two weeks would suffice.

Regarding the questions about documentation, it became evident that the longer people work on KIELER, the lower their opinion about KIELER's documentation quality became. Nearly all of the developers stated that the documentation should be improved. This could be done by giving more and larger examples in the wiki, by writing more verbose comments in the code, and by improving the wiki's structure. Developers agreed that documentation quality varies heavily between the different parts of KIELER. Also some developers said that a better project wiki would encourage them to do more documentation work there. To avoid having too many outdated wiki pages, one developer suggested that there should be some kind of expiration date for wiki pages, so that the respective developers could be forced to revise pages that pass this date.

Tickets and a bug tracking system are considered as fine tools for maintaining communication among developers and for issue tracking. Most developers only use tickets to file bug reports for other developers. Only few developers are using tickets to maintain their own task list. One developer strongly suggested that more developers should use ticketing as a task tracking tool. This would make it easier to get an idea of what other people are working on. The usability of the bug tracking tool currently used was rated as rather bad. The ticket work flow is seen as rather confusing, and querying for a subset of tickets does not work well. Some developers would appreciate an Eclipse integration for ticketing, while others prefer separate applications, the way it currently is.

All developers agreed that regarding the release process many things need to be improved. For many developers the process is unclear. There is a loose schedule of half-yearly releases. After a release date is fixed, many new bugs are discovered as it seems that thorough testing only occurs after scheduling a release. Thus releases are always delayed at least one month. Additionally most developers are unable to estimate how much work is yet to be done until a release is ready to be made. People agree that a fixed and more strict release schedule with a clear sequence of steps such as feature freezes could be a good countermeasure. Also releases are often delayed, because some people insist that a certain feature should be included, which is not ready to be released yet.

The situation with build management is described similarly. While most developers know that there is a server that does automatic nightly builds, nobody except the developer that implemented the build server could describe how the build process actually works. One developer did not even know that there is a web page showing the build status. Other

developers say that the only thing they know about the process is that they sometimes receive mails about failed builds. Even the developer responsible for the build said that it is rather messy, with many hacks included to make things work.

The last part of the interview deals with questions about the code itself, how code is tested, how people work with source code management, and how code quality could be improved. KIELER has recently moved its sources from Subversion to the distributed revision control system Git. Some developers are still afraid of Git's complexity, while most people discovered, after a somewhat steep learning curve, working with Git as quite enjoyable. Depending on their experience with Git, developers use advanced features such as branches, stash, and server-side clones.

The overall code quality of KIELER was described as mixed. While the core parts are well written and well documented, there are some parts that are ugly. This often depends on which developer has written the code. All developers agreed that KIELER coding and user interface conventions are satisfied for most parts. Usage of style checkers such as *CheckStyle* is widespread between developers. However, hardly anyone uses tools for static code analysis such as *FindBugs* to check their code. All developers stated that they rework their code regularly, at the latest when implementing new features. Regarding software tests the interviewed have made clear that automatic testing is not done in KIELER. Developers explained that they test their own code with a personal set of examples to see whether it works. When asked how the developers determine that their changes do not interfere with other parts of KIELER answers varied from 'testing if it compiles' to 'I think I know whether my changes might break other parts'. All developers have been confident that the introduction of unit tests would lead to a more stable codebase.

When asked about code and design reviews, all developers agreed that reviews are a proper method for improving the overall code quality, and that they see personal benefits in such reviews as well. However, reviews occur only rarely. Most developers know that there is a review process and deemed it too heavy-weight and complex. Code reviews tend to require too much time as they are often not properly prepared by reviewers and the code authors, leading to reluctance against reviews. However, all developers said that if there was a lean and structured review process, they would be eager to do more reviews as they see that more review sessions are needed.

4.2 Developers in other Academic Projects

Other academic projects might face the same challenges as KIELER. To find out about the situation in other academic projects, the project lead of the *Kieker* project was interviewed. *Kieker* is a tool for analysing software run-time behaviour. It is used to collect and visualise runtime data. *Kieker* is written in Java and AspectJ and is licensed under the Apache License. Currently there are six staff members working on the development of *Kieker*, additionally there are about six students who utilise the *Kieker* libraries to analyse software. The questions asked were similar to those for the KIELER developers, however some KIELER specific questions were left out.

The first part of the interview dealt with project organisation. Communication in the team works well. There are monthly meetings. About one and a half weeks before the meeting, a call for topics mail is issued, and topics are collected in *Kieker's* project management wiki. Special topics are discussed in additional smaller meetings. However, the project lead would like to see more contributions by other team members to the meeting's agenda. Another large

part of communication is handled via the web-based project management software *Trac*. It is used for bug-tracking, as a documentation wiki, and also as a tool for collaborative writing, for example to fix meeting agendas. Finally, *Trac* is used for milestone planning. The developers are satisfied with *Trac's* features.

Project documentation is maintained in both a user manual and a development guide. The user manual is well written and complete, while the development guide lacks detail in some parts. To achieve good documentation quality a student assistant is employed from time to time to rewrite and update certain sections of the manual. Documentation is also checked and updated as part of the release process.

Kieker uses a fixed half-yearly release cycle. Four weeks before a release all open tickets are examined and classified as relevant or irrelevant for the current release. Irrelevant tickets are postponed for the next release. All open tickets scheduled for the current release are then reassigned to the responsible developers and a deadline for open tickets is fixed. Three weeks before the release the API is frozen. The user guide has to be updated until two weeks before the release. The last week before the release is reserved for last tests and bugfixes. This rather strict schedule helped to carry out punctual releases at fixed dates and works well for *Kieker*.

The second part of the interview was concerned with *Kieker's* code base and asked how high code quality is maintained. Source code management is done in Git, new features are developed on branches to keep the master branch stable. Unit tests are developed parallel to new features to achieve high test coverage and to prevent regressions. After an external code review the usage of static code checkers such as *CheckStyle*, *FindBugs*, and *PMD* has been made mandatory. *Kieker* is built via *ant* scripts. There is a continuous integration server conducting hourly builds, including static code analysis and unit tests. Code reviews are only done occasionally but are seen as a useful method of maintaining high code quality.

4.3 Commercial Developers

To find out about how commercial software development is done and how KIELER could benefit two interviews with software developers, one working for a software company that develops a web application in the social media sector, and the other working on software for port logistics were conducted. The main objective of these interviews was to find out about how things in a company are run different from an academic research project. The interviews were both done as informal dialogues. Learning from the interviews it could be observed that even though development is done differently in different companies there are certain similarities. Both developers work in small software teams. Both teams employ well-defined and well-described development processes, one *SCRUM* and one *Feature Driven Development*. These Processes will be described in more detail in chapter 3 when also the application of those on the development of KIELER will be discussed.

4.4 Discussion

When comparing commercial and academic software development and even academic projects with each other, many differences become evident. Looking at commercial projects, there is nearly always the objective of releasing a finished product for a customer. This is different to KIELER where even the project's aims are part of the research. Furthermore in companies there is often a whole team working on a certain part of software. In KIELER, development is

mostly done as part of theses, with only one person working on a certain feature. KIELER is developed by a group of individuals, different developers might even pursue different goals with KIELER depending on their fields of research. Another important point is that most KIELER developers are not employed for the sole purpose of software development. They are researchers or students and programming is only part of their research activity. There are times when much development takes place, and other times when individual developers have not written even a single line of code for weeks. Learning from the interviews, one should consider exploring if and how well-defined processes used in commercial software development are applicable for KIELER.

In comparison with *Kieker* there are also many differences to KIELER. While in *Kieker* development is mostly done by staff members, and students use the *Kieker* libraries to develop applications for their theses, in KIELER students work on the software itself. This is mostly due to the structure of KIELER. As an Eclipse RCA KIELER is comprised of about 150 Eclipse plugins, a common task for a thesis is to implement a special plugin or demonstrator. However, the advice to use more static code checkers and to implement a more strict release schedule could be beneficial for KIELER.

Looking at the interviews with the KIELER developers it becomes very clear which parts of the project's organisation are in severe need of improvement. The build and the release process will have to be restructured in a more intuitive and transparent way. It is very important for all developers to know about how things are done. There will have to be more code and design reviews in a more light-weight process. Usage of the ticket system has to be improved and ticket workflow has to be monitored more closely. Developers need to be encouraged to write unit tests to raise test coverage and limit the number of regressions. Overall we will need a process that is not too time-consuming.

Chapter 5

Building KIELER

When software projects grow larger, there is also a growth in complexity when it comes to compiling and building the final product. In small projects with only a small number of files it may be viable to call the compiler by hand. Larger projects however, consist of many modules that depend on each other so that build order becomes important. This makes building large projects by hand too complicated to be feasible.

While the KIELER RCA can be manually built from the Eclipse IDE by using the RCP export wizard, having this as the only way to build has many drawbacks. When building by hand, it is for example impossible to provide regular nightly builds for software testers. Furthermore, there is no reference build platform as all developers build on their own installations which might differ from each other. To ensure a stable code base it is desirable to do as many builds as possible, for example after each source code checkin. This helps to find problems early and ensures that the work of a single developer does not interfere unexpectedly with other parts of the software. It also allows to run unit tests regularly. In order to do automatic builds after each source code check in, an option to do headless builds has to be implemented. This chapter first describes how headless builds are currently done in KIELER and lists the problems of this process. We then discuss other ways of how automatic builds in Eclipse can be done. Afterwards the implementation of a new headless build process using the build and project management tool *Maven* is described.

5.1 Headless Builds of Eclipse-Based Applications

There are basically three common ways for building eclipse plugins, features, and rich client applications. The first way is to build manually within Eclipse using the RCP export wizard. The second way is to use the *PDE build* scripts together with the build tool *Ant*. The third way is to use the automation tool *Maven* together with *Tycho*, a set of plugins to make Eclipse metadata available to Maven.

5.1.1 The current Build Process

Currently KIELER is built with Ant and PDE build. Ant is a software tool for automating software builds of Java projects. The Ant build process is described in a monolithic XML file that is by default named `build.xml`. Therein it is possible to specify certain build *targets* in which build tasks are performed. The following example illustrates a simple Ant target:

```
<target name="compile" description="compile .java source files to .class files">
  <mkdir dir="classes"/>
  <javac srcdir="." destdir="classes"/>
</target>
```

In the example above a target `compile` is defined. The compile target executes two tasks, `mkdir` and `javac`. `mkdir` creates a directory on the filesystem and `javac` calls the Java compiler to compile java source files into Java bytecode. Ant provides a set of built-in tasks such as `mkdir` and `javac`. Furthermore, Ant can be extended by so called *antlibs*.

The *Eclipse Plugin Development Environment* (PDE) provides tools to create, develop, test, debug, build, and deploy Eclipse plugins, features, and RCP products. Part of the PDE is *PDE build* which facilitates the automation of build processes. With PDE build it is possible to produce Ant scripts to build Eclipse plugins, features, and RCP products.

The current KIELER headless build process uses PDE build to generate Ant build files to build KIELER's plugins, features, and the KIELER RCA. The generation process, and therefore PDE build, is controlled by a master `build.xml` file. However, the documentation of PDE build is in severe need of improvement. There is nearly no written documentation to be found; the information about how to write a proper `build.xml` file has to be gathered from example files. Furthermore, PDE build's functionality is very limited. While it is straightforward to generate build files for single plugins and small projects, using PDE build for projects as large as KIELER leads to a very complicated and large master `build.xml`. This is due to the fact that KIELER integrates many third-party technologies, such as code generation with *Xtend*, that have to be integrated manually. Currently, the only person who understands the build process on the whole is the developer that implemented it. During the interviews he stated that he had to implement many hacks to make things work and to circumvent the limited functionality of PDE build. Automatic Javadoc generation or static code checking for example has to be implemented manually in `build.xml`.

Another part of the current KIELER build process are nightly builds. To conduct those nightly builds automatically, the popular continuous integration tool *Hudson* is used. Hudson is written in Java and runs in a servlet container, for example *Apache Tomcat*. Hudson can be configured to execute Ant based projects as well as arbitrary shell scripts. It also supports the distributed source code management tool git so it can be configured to automatically checkout sources from a source repository. To build KIELER, Hudson executes the following tasks every night:

1. Checkout the current KIELER sources.
2. Build the KIELER RCA and KIELER P2 repository.
3. Deploy the build artifacts to the local webserver for public download.

Furthermore there is a nightly task for Javadoc generation defined in Hudson. In case of build failures Hudson notifies the developers about the failed build and also stores the build log so people can find the reason for the failed build.

5.1.2 Introducing Maven

The *Maven*[1] project aims at providing a management framework for Java projects. This means that Maven can not only build software but can also do many more things, such as generating a project website, conduct releases, maintain documentation, connect to source code management systems, etc. [CMPS06]. This part will focus on building software with maven.

One important thing to notice about Maven is its modular structure. The basic Maven installation contains only the shell script `mvn` and a small set of `jar`-files providing the Maven

[1] `http://maven.apache.org`

core and basic configuration. Most of Maven's functionality is implemented in plugins. Plugins are dynamically downloaded as needed from central repositories, which are either specified within the build-specific Maven configuration or in the central Maven configuration.

Maven defines a standard lifecycle for building, testing, and deploying project artifacts. A Maven project is described by a hierarchical *Project Object Model (POM)* that is implemented in XML files. The default Maven lifecycle is made up of 23 phases including the following:

validate validates the project's correctness and tests if all necessary information is provided

generate-sources generate any sources for inclusion in compilation

compile compiles the project's sources

test tests the compiled code using a unit test framework

package packages the code into *JAR* files

integration-test runs integration tests in the runtime environment of the package

In each of these phases certain Maven plugins that implement different goals are run. The *maven-compiler-plugin* for example provides a goal `compile:compile`, which is run by default in the *compile* phase. Other goals include the generation of Javadoc in the goal `javadoc:aggregate`. The first part of a goal's name is the namespace of a certain plugin while the second part defines an action. Goals may be run in three different ways: as a default part of a phase, explicitly on the command line, or attached to a phase in a `pom.xml` file. The most basic `pom.xml` looks like the following:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.namespace.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

When Maven is run with this simple `pom.xml` file it works off the default lifecycle, running the default phases with the default plugins and their default goals. This is only sufficient for small projects however; as projects grow larger more configuration is required.

Maven is also able to handle so called multi-module projects. Multi-module projects contain several sub-projects that share a common parent `pom.xml`. Submodules inherit all settings from their parent `pom.xml` file while the parent POM file contains references to all submodules. This allows the definition of build dependencies in the submodule's POMs which are automatically resolved by Maven.

As we have learnt, a Maven build cycle is divided into certain phases. Within these phases different plugins that implement different goals are executed. When a plugin is used for the first time, it is fetched from a central maven repository and stored in a local repository. On Linux machines the local repository can be found under `/home/<user>/.m2`. Configuration for the different plugins is done in the POM files. The following example illustrates how Javadoc generation is configured:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>2.8.1</version>
  <configuration>
```

```
      <doclet>de.cau.cs.kieler.doclets.RatingDoclet</doclet>
      <docletArtifact>
        <groupId>de.cau.cs.kieler</groupId>
        <artifactId>de.cau.cs.kieler.doclets</artifactId>
        <version>0.0.1-SNAPSHOT</version>
      </docletArtifact>
      <useStandardDocletOptions>false</useStandardDocletOptions>
    </configuration>

    <executions>
      <execution>
        <id>aggregate</id>
        <goals>
          <goal>aggregate</goal>
        </goals>
        <phase>integration-test</phase>
      </execution>
    </executions>
  </plugin>
```

The first part of the XML snippet describes which plugin is to be configured; in this case it is the *maven-javadoc-plugin*. In the `<configuration>` section, plugin-specific settings are configured. Finally, the execution section specifies which of the plugin's goals should be run in which phase. In this case the goal *aggregate* is to be run in the *integration-test* phase.

5.1.3 Maven and Eclipse

While Maven provides the necessary plugins to compile standard Java applications out of the box, compiling and packaging Eclipse plugins, features and rich client applications remains very complicated. Compilation is complicated due to the huge amount of dependencies between the 150 KIELER Eclipse plugins and even more due to dependencies to other third party plugins and the Eclipse runtime. Furthermore the standard Maven packaging plugin does not know how to package Eclipse plugins, features and rich client applications.

The *Eclipse Tycho project*[2] provides a set of Maven plugins to solve these problems. The Eclipse IDE stores metadata containing dependency information in a file called *MANIFEST.MF* for each Eclipse plugin. A MANIFEST.MF file is a set of key-value pairs:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: KIELER Layered Layouter
Bundle-SymbolicName: de.cau.cs.kieler.klay.layered;singleton:=true
Bundle-Version: 0.2.0.qualifier
Bundle-Vendor: Christian-Albrechts-Universitaet zu Kiel
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Require-Bundle: com.google.guava;bundle-version="9.0.0",
 de.cau.cs.kieler.core,
 de.cau.cs.kieler.core.kgraph,
```

[2] http://www.eclipse.org/tycho

```
de.cau.cs.kieler.kiml
```

The Tycho Maven plugin makes this information available to Maven. All dependencies are listed in the `Require-bundle` key. This enables Maven to determine a build order for all KIELER plugins and features. Furthermore, the *tycho-packaging-plugin* introduces new packaging types for Maven including *eclipse-plugin*, *eclipse-feature*, and *eclipse-application*.

Enabling Tycho for a Maven project is straightforward, as for any Maven plugin some lines have to be added to `pom.xml`:

```
<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>tycho-maven-plugin</artifactId>
  <version>0.15.0</version>
  <extensions>true</extensions>
</plugin>
```

This tells Maven to load the Tycho plugin. Afterwards, Maven is able to build an Eclipse plugin by setting the packaging type to *eclipse-plugin* in the plugin's POM file.

Another important thing is automated unit testing. Unit tests in eclipse are implemented as special test plugins. From the Eclipse IDE they are run as a *JUnit Plugin Test*. The *tycho-surefire-plugin* is able to run these plugin tests headlessly by providing an Eclipse runtime for the tests to be run in. This is enabled by setting the plugin's packaging type to *eclipse-test-plugin*.

5.1.4 Choosing a Build System for KIELER

Looking at the KIELER build process basically leaves two choices. As the current build with PDE Build has become obscure over the years and is also full of hacks to circumvent certain problems with PDE build, the choices are either to re-implement the whole process with PDE Build in a clean way or to set up a new process from scratch using Maven and Tycho.

In the Eclipse community there is a commitment to migrate all official Eclipse projects to a common build infrastructure. This common build infrastructure lists Maven/Tycho as the preferred build technology. Furthermore, the Maven approach with its hierarchic structure of POM files allows a more transparent approach to configure plugin-specific build options as compared to a central Ant `build.xml` file. Also, due to the large amount of Maven plugins, it is very simple to implement more build-specific tasks like automatic static code checking, unit tests, publishing of build artifacts, etc. As Maven/Tycho provides far more benefits compared to PDE Build and is designated to be the standard build tool for the Eclipse platform, Maven is chosen as the new build system for KIELER.

5.2 Implementing a KIELER-build with Maven and Tycho

On the way to a KIELER build with Maven and Tycho, several steps have to be taken and several decisions have to be made. Also, there are things to be considered because of their impact on things, which might or might not lead to certain positive, neutral, or negative consequences. First of all, Maven has to be installed on the build machines. In order to have a transparent and clear build, one must needed to think about the structure of the POM files carefully. After the core of the build process is working, i. e., the KIELER RCA builds with Maven, additional tools for quality assurance such as static code checking and unit tests have

to be added to the build. Finally, the build has to be implemented on a *continuous integration* server in order to do automatic builds after each code checkin and automatic nightly RCA builds.

5.2.1 Installing Maven

Installing Maven is very straightforward. As most of the Maven functionality is implemented in plugins which are downloaded at runtime, the installation archive is rather small. After downloading the `.tar.gz` archive is extracted to `/home/java/maven`. Afterwards some environment variables have to be set. The file `/home/java/java-env` already contains environment settings for the Java installation. The following lines are added:

```
M2_HOME=/home/java/maven
export MAVEN_OPTS="-Xmx2048M -XX:MaxPermSize=256M"
M2_BINDIR=$M2_HOME/bin
PATH=$JAVA_BINDIR:$M2_BINDIR:$PATH
```

This adds the location of the Maven binary to the executable search path. As building software tends to be very memory-consuming, the Java Virtual Machine's heap memory is set to 2048 megabytes through the variable `MAVEN_OPTS`. Afterwards Maven may be use by running `mvn` on the command line. Further settings are configured in the global `settings.xml` file which can be found in `/home/java/maven/conf`. Most settings there may safely be left at their defaults. However, to speed up downloading of plugins, an HTTP proxy server is added:

```
<proxy>
  <id>optional</id>
  <active>true</active>
  <protocol>http</protocol>
  <host>www-cache.informatik.uni-kiel.de</host>
  <port>3128</port>
  <nonProxyHosts>rtsys.informatik.uni-kiel.de</nonProxyHosts>
</proxy>
```

This configures the institute's *Squid-Proxy* to be used for all HTTP requests with the local webserver as exception. This completes the installation of maven. Afterwards Maven can be used by sourcing `java-env` and running `mvn` on the command line.

5.2.2 Structuring the Build

As we have learnt before, POM files can be structured in a hierarchical way. This means that settings can be inherited over multiple generations of parent POM files where global settings are defined. As a consequence we need to consider how the POM files are to be structured. Looking at the KIELER source tree reveals the following structure:

```
/
|---/plugins
|   |---/de.cau.cs.kieler.plugin_1
|     ...
|   |---/de.cau.cs.kieler.plugin_n
|---/features
```

```
|---/de.cau.cs.kieler.feature_1
  ...
|---/de.cau.cs.kieler.feature_n
```

KIELER's plugins are located as subfolders in the `plugins` folder, while all features are to be found as subfolders in the `features` folder. To make things as understandable as possible, the POM hierarchy should somehow respect the filesystem hierarchy. Furthermore each single plugin and feature needs to be compiled and packaged separately and therefore needs its own POM file. This leads to a structure in which all single plugin POM files inherit settings from a common parent POM file for all plugins. The same holds for features, each feature's POM file inherits from a common parent file for features. Finally the common POM files for features and plugins inherit project wide settings from a common parent POM file. Furthermore, a special repository project has to be added for assembling the KIELER RCA and the KIELER P2 repository.

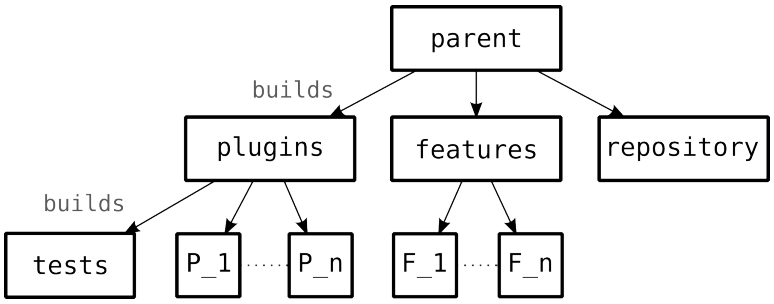


Figure 5.1. KIELER build hierarchy

5.2.3 The parent POM

The parent POM file contains all settings common for the whole build. The first part of a POM file describes a namespace, the artifact id, version, and packaging:

```
<groupId>de.cau.cs.kieler</groupId>
<artifactId>parent</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>pom</packaging>
```

The artifact id indicates that this is the parent POM file for the `de.cau.cs.kieler` group. As the KIELER parent POM file does not actually produce compiled Java artifacts, but contains only metadata, the packaging type is set to `pom`.

The `<properties>` section in the parent POM contains some global settings for the version of the Tycho plugins and the target JDK:

```
<properties>
  <tycho-version>0.15.0</tycho-version>
  <targetJdk>1.5</targetJdk>
</properties>
```

In order to be compatible with Eclipse the target JDK is set to 1.5.

The <modules> section in the parent POM file determines which children are to be compiled in the build reactor:

```
<modules>
  <module>../../plugins</module>
  <module>../../features</module>
  <module>../de.cau.cs.kieler.repository</module>
</modules>
```

As described before, there are two intermediate POM files that compile plugins and assemble features. Furthermore a special de.cau.cs.kieler.repository module is introduced. This module is responsible for building the KIELER RCA and the P2 repository and will be described in more detail later.

Eclipse plugins, features, and rich client applications are built against a certain Eclipse target platform that provides all dependencies. If an Eclipse plugin is compiled interactively in Eclipse, the Eclipse installation is usually used as the target platform and thus provides all dependencies. For Maven/Tycho builds however, dependencies have to be stored in Eclipse P2 repositories. As the headless KIELER build should use the central Eclipse installation of the Rtsys Group, a P2 repository is generated from the installation and stored in /home/java/repository/juno382:

```
java -jar eclipse_3.8/plugins/org.eclipse.equinox.launcher_*.jar \
     -application org.eclipse.equinox.p2.publisher.FeaturesAndBundlesPublisher \
     -metadataRepository file:/home/java/repository/juno382 \
     -artifactRepository file:/home/java/repository/juno382 \
     -source /home/java/eclipse_3.8/ \
     -publishArtifacts
```

Afterwards the generated repository is made available to Maven:

```
<repository>
  <id>p2.juno38</id>
  <layout>p2</layout>
  <url>file:/home/java/repository/juno382/</url>
</repository>
```

Unfortunately, as the Eclipse reference installation is on the Linux platform, the generated repository does not include platform specific fragments for other platforms such as Windows or Mac OS. Therefore the remote central Eclipse P2 repository available at http://download.eclipse.org/releases/juno is also added to the build for Maven to be able to solve as build dependencies.

In the <plugins> section of the parent POM file, all Maven plugins that are used and that need special configuration have to be configured. For the first approach of a basic KIELER RCA build the following plugins are configured.

tycho-maven-plugin Enables Tycho and thus makes Eclipse metadata available to Maven

tycho-source-plugin Enables the packaging of plugin sources for developers

xtend-maven-plugin Runs the Xtend standalone compiler on Xtend sources to generate Java code.

build-helper-maven-plugin Adds directories that contain generated sources to the build

maven-clean-plugin Cleans the source tree by deleting old build artifacts; needs to be configured to also clean up generated sources

target-platform-configuration Specification of target operating system platforms for multi-platform builds

This is the basic set of plugins needed to build KIELER. Further plugins will be introduced later when static code checking and unit tests are described.

The plugins and features POM file

The POM file in the plugins subdirectory basically lists all plugins that are to be built as modules. All configuration is inherited from the global parent POM:

```
<parent>
  <groupId>de.cau.cs.kieler</groupId>
  <artifactId>parent</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <relativePath>../build/de.cau.cs.kieler.parent/pom.xml</relativePath>
</parent>
```

Afterwards all plugins are listed as modules:

```
<modules>
  <module>../test</module>
  <module>de.cau.cs.kieler.core</module>
  <module>de.cau.cs.kieler.core.annotations</module>
  .
  .
</modules>
```

The module list also contains a special module *test* that is needed to compile and run all unit tests. The main reason for introducing an intermediate POM file for plugins is to be able to build all plugins and tests separately from the RCA and from the features. If Maven is called in the `plugins` subdirectory, only plugins will be built and tests will be run. This speeds up build time for continuous integration builds which will be described later.

Finally, a POM file for each single plugin needs to be written. Fortunately these POM files are very short as nearly all configuration was done in the parent POM files. First of all each plugin needs to include the plugins parent POM file:

```
<parent>
  <groupId>de.cau.cs.kieler</groupId>
  <artifactId>plugins</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</parent>
```

This is common to all plugins. Afterwards, each plugin's artifact ID, version, and packaging type has to be specified:

```
<groupId>de.cau.cs.kieler</groupId>
<artifactId>de.cau.cs.kieler.core</artifactId>
<version>0.7.0-SNAPSHOT</version>
<packaging>eclipse-plugin</packaging>
```

The parameters `groupId` and `packaging` are common to all of KIELER's plugins. Only the `artifactId` and `version` are specific to each plugin. Since all single plugin POM files are very similar, it is simple to automate POM generation, for example with a *PERL* script. However, some plugins contain *Xtend* sources or platform-specific fragments. In this case specific configuration has to be written for the corresponding plugins.

In order to run the standalone Xtend compiler for source generation during compile time, the following has to be added to the corresponding POM file:

```
<plugin>
  <groupId>org.eclipse.xtend</groupId>
  <artifactId>xtend-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

This enables the *compile* goal of the *xtend-maven-plugin*.

In case of platform specific fragments the *tycho-target-platform* plugin has to be configured accordingly:

```
<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>target-platform-configuration</artifactId>
  <version>${tycho-version}</version>
  <configuration>
    <environments>
      <environment>
        <os>win32</os>
        <ws>win32</ws>
        <arch>x86</arch>
      </environment>
    </environments>
  </configuration>
</plugin>
```

This specifies that the plugin contains specific fragments for the *win32* operating system on the *x86* architecture. Other examples of operating systems include *Linux* and *macosx*. Possible architectures include *x86*, *x86_64* and *sparc*.

The structure of the features POM file is basically the same as with the plugins POM file. Configuration is imported from the global parent POM file and all features are listed as modules. Also, each feature is accompanied by a source feature that contains the corresponding source plugins. Developers that want to implement software on top of a KIELER feature may also install the corresponding source feature. This way the Eclipse IDE can provide further information on KIELER's classes and methods.

Each feature also needs its own POM file. Fortunately, similar to the plugin POM files, these are also very simple as all configuration is inherited from parent POM files. Generation

of these POM files is done the same way as for plugin POMs, with the exception that the correct packaging type for features is `eclipse-feature`.

The RCA and repository project

This project is needed to build the KIELER RCA and the KIELER P2 repository. As an Eclipse RCA is basically a P2 Repository together with startup scripts, branding, and the Eclipse runtime, the proper packaging type to configure in the project's POM file is `eclipse-repository`:

```
<version>0.6.0-SNAPSHOT</version>
<artifactId>de.cau.cs.kieler.repository</artifactId>
<packaging>eclipse-repository</packaging>
<name>KIELER Repository</name>
```

This configuration is sufficient to build a P2 repository. In order to also assemble an RCA some further configuration in the `<build>` section is needed:

```
<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>tycho-p2-director-plugin</artifactId>
  <version>${tycho-version}</version>
  <executions>
    <execution>
      <id>materialize-products</id>
      <goals>
        <goal>materialize-products</goal>
      </goals>
    </execution>
    <execution>
      <id>archive-products</id>
      <goals>
        <goal>archive-products</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

These settings instruct Maven to load the `tycho-p2-director-plugin` to execute afterwards the `materialize-products` and `archive-products` goals in their default phase. The materialisation goal reads all `.product` files in the current directory and produces Eclipse RCP applications according to the settings in the product files. The File `kieler.product` contains all the necessary information to build the KIELER RCA. In the first part of a product file the product's branding such as names, icons, launcher arguments, etc. are specified. It also contains the full license text, in this case the *Eclipse Public License*. Later, in the `<features>` section, all features to be to be included in the RCA are listed.

Finally, the `archive-products` goal compresses all materialised products into `.zip` files for later deployment on the KIELER download site.

5.2.4 Unit Tests

In order to maintain a stable build and prevent regressions from occurring, unit tests are added to the build. For Eclipse plugins, tests are implemented in special test plugins that contain unit tests to be run in the Eclipse runtime. All test plugins are placed in the `test` subdirectory. Similar to plugins and features the `test` subdirectory contains an intermediate POM file that lists all test plugins to be compiled and run as modules. Each test plugin directory contains a POM file whose packaging type is set to `eclipse-test-plugin`. This instructs Maven to compile the plugin and run the *tycho-surefire-plugin* afterwards. This plugin runs the test plugins as *JUnit* tests in the Eclipse runtime. After running the tests, the test results can be found both in text as well as in machine readable XML files. A sample test output looks like this:

```
-------------------------------------------------------------------------------
Test set: de.cau.cs.kieler.core.test.math.BezierSplineTest
-------------------------------------------------------------------------------
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 sec
testGetStartPoint(de.cau.cs.kieler.core.test.math.BezierSplineTest)  Time: 0.001
testGetEndPoint(de.cau.cs.kieler.core.test.math.BezierSplineTest)  Time: 0
testGetInnerPoints(de.cau.cs.kieler.core.test.math.BezierSplineTest)  Time: 0.001
testGetBasePoints(de.cau.cs.kieler.core.test.math.BezierSplineTest)  Time: 0
testGetPolylineApprx(de.cau.cs.kieler.core.test.math.BezierSplineTest)  Time: 0.001
```

This is the output of the *BezierSplineTest*. This particular test contains five test methods that were all successful.

When running tests on KIELER plugins depending on the Eclipse user interface, the `tycho-surefire-plugin` needs some more configuration in the test plugin's POM file.

```
<plugins>
  <plugin>
    <groupId>org.eclipse.tycho</groupId>
    <artifactId>tycho-surefire-plugin</artifactId>
    <version>${tycho-version}</version>
    <configuration>
      <useUIHarness>true</useUIHarness>
      <useUIThread>true</useUIThread>
    </configuration>
  </plugin>
</plugins>
```

When running these tests from the command line, an Eclipse window is displayed during testing. On a headless build server, normally there is no X server to connect to. In order to provide a virtual X server the virtual X frame buffer *Xvfb* has to be started:

```
$ Xvfb -extension RANDR :75 &
```

This enables the build process to connect to a virtual X server running on display :75.

5.2.5 Static Code Checking

Maven's plugin-based architecture allows to add third party software for static code checking easily. The KIELER project guidelines require the usage of *Checkstyle*[3], a static code analysis tool for checking if Java code complies to a given set of coding rules. The KIELER coding rules are specified as XML in the file `checks.xml`. In order to check whether all developers use Checkstyle for their code, the *maven-checkstyle-plugin* is added to the KIELER parent POM file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>2.9.1</version>
  <configuration>
    <configLocation>checks.xml</configLocation>
    <sourceDirectory>src</sourceDirectory>
  </configuration>
</plugin>
```

This loads the *maven-checkstyle-plugin* with `checks.xml` as rule file. Furthermore, style checking is restricted to the `src` subdirectory. This prevents Checkstyle to be run on generated code usually located in in folders named `src-gen` and `xtend-gen`. As it might have already been noticed, the plugin configuration for Checkstyle does not contain a `<goals>` section. This means the checkstyle goals are not run by default, but only when the `checkstyle:checkstyle` goal is specified explicitly on the command line. This speeds up build times, but also leaves the possibility to implement a special quality assurance build for all static code checking later.

Another code analysis tool used in KIELER is *FindBugs*[4]. In contrast to Checkstyle, FindBugs does static code analysis on Java bytecode, i. e., on `.class` files. FindBugs tries to detect potential runtime problems in Java programs by searching for common bug patterns in the bytecode. FindBugs is added to the KIELER build in the parent POM file as follows:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>findbugs-maven-plugin</artifactId>
  <version>2.4.0</version>
  <configuration>
    <xmlOutput>true</xmlOutput>
    <failOnError>false</failOnError>
    <maxHeap>2048</maxHeap>
  </configuration>
</plugin>
```

As FindBugs tends to be very memory-consuming, the maximum heap space for FindBugs is set to 2048 megabytes. As with Checkstyle, in order to run FindBugs the `findbugs:findbugs` goal has to be explicitly specified on the command line.

[3] `http://checkstyle.sourceforge.net/`

[4] `http://findbugs.sourceforge.net`

5.3 Continuous Integration

Continuous integration (CI) is a technique in software engineering that aims at improving software quality and at reducing delivery time by applying quality control continuously after each source code checkin, rather than before a release takes place [DMG07]. Principles of CI include fast, automatic, self-testing builds. It is mandatory that all code changes have to be committed regularly and new features have to be implemented in small steps that do not break the build. An optimal CI schedule would implement the following principles:

1. Maintain a code repository
2. Automate the build
3. Make the build self-testing
4. Make every commit trigger the automatic build
5. All developers should be informed about current build status
6. Provide access to build artifacts
7. Automate deployment

The first item has already been implemented, as KIELER's sources are managed with Git. This section deals with the implementation of items two to seven. First of all, a suitable software to help conducting CI has to be chosen. Afterwards configuration of automatic KIELER builds is described.

5.3.1 Hardware Considerations

Currently the KIELER sources are kept on a 19" 2-unit Rack server manufactured by Supermicro. The machine contains the following hardware:

- Supermicro X8DTN+-F dual Xeon Mainboard
- Two Intel Xeon E5506 CPUs running at 2.13GHz
- Six 4GB DDR3 memory modules
- Three 320GB hard disks configured as RAID-1 array with one hot spare disk

As source code management does not put much strain on the CPUs, this machine is suitable to also host the KIELER build infrastructure. The machine has eight CPU cores, which allows parallel building of multiple build jobs if many code checkins occur in short intervals. Furthermore, the machine provides enough resources for further project management tools for KIELER.

5.3.2 Software Candidates

There are many software tools that help to implement continuous integration for software projects. A software tool that is suitable for the KIELER project has to meet certain requirements. First and foremost, the tool needs to run on the Linux platform as the designated build machine is running Ubuntu Linux 12.04. Additionally, the software has to provide means to compile, test, and package Java projects using Maven. Further requirements include the possibility to execute arbitrary shell scripts for automatic deployment, the ability to gather

XML data from Checkstyle and FindBugs, and a clean and understandable web interface to display build results, statistics, etc. In the following, two suitable candidates are presented.

Hudson

Hudson is a continuous integration tool originally developed by Kohsuke Kawaguchi at Sun Microsystems. Hudson is released as open source software under the MIT license. Recently, after Sun Microsystems has been taken over by Oracle, the project lead has passed to the Eclipse Foundation. The current production version is 2.2.1, released in May 2012. The first release by the Eclipse Foundation will be version 3.0.0 which is currently in beta.

Hudson satisfies all requirements needed in the KIELER project. It can execute Maven builds and there are many plugins providing further functionality, such as the collection of code analysis results and automatic deployment. The Eclipse Common Build Initiative lists the upcoming 3.0 release as the preferred CI tool for the Eclipse community. Currently version 2.1.2 is running on `http://hudson.eclipse.org` for building Eclipse projects.

However, Hudson also has its limitations. As the KIELER developers have stated, Hudson's web interface is rather confusing and non-intuitive. For example, successful builds are indicated by blue bubbles which is uncommon compared to red/green indicators. The configuration interface is similarly difficult to understand due to unneeded complexity. Furthermore, Hudson's documentation is very sparse, there are only some wiki pages providing mainly information on installing Hudson on different platforms.

Hudson is distributed as a Java web archive file. This means it needs to be deployed in a Java servlet container such as *Apache Tomcat*, which has to be configured separately. This means greater effort in configuration and also in administration as two different pieces of software have to be maintained separately.

Bamboo

Bamboo is a continuous integration server developed by Atlassian Software[5]. Bamboo is commercial software with a licensing model that provides a perpetual license with twelve months of free product updates and costs based on the number of concurrent build jobs needed. However, Atlassian provides free licenses for open source projects meeting the following requirements:

- has a publicly available website
- has an approved open source license
- has an established code base
- has at least one release

As KIELER meets all of the above requirements it is entitled to apply for such a free license.

The current Bamboo production release is 4.1.2. Bamboo is distributed in various configurations. It is possible to download Bamboo either as a bare Java web archive or as a standalone .tar.gz archive that includes everything needed to run the software. Bamboo comes with a clean web interface. A configurable Dashboard lists build status for all projects. Successful builds are indicated by a green circle with a checkmark, while failed builds are indicated by a red circle with an exclamation mark. Bamboo supports building Maven projects and also fulfils the other requirements for a KIELER CI tool. Bamboo comes with extensive online

[5] `http://www.atlassian.com`

documentation containing an administrator and a user manual which describes the usage and installation of the software.

Discussion

Both of the presented continuous integration tools meet the requirements for conducting automatic builds of KIELER. As KIELER is an open source tool it would be in the spirit of open source to also use an open source tool for continuous integration. However, Bamboo's clean user interface provides many advantages compared to Hudson. Furthermore, the KIELER developers stated in the interviews that they have problems understanding the information Hudson provides. Feature-wise, Bamboo is clearly the best suited tool for continuous integration in KIELER. One drawback of Bamboo is its commercial license. Atlassian provides free licenses to open source projects, but there is no guarantee that someday politics might change. However, this possibility is rather unlikely as Atlassian's open source licensing hast not changed since its founding in 2002. The most serious argument for preferring Bamboo over Hudson is that during a short evaluation, developers liked Bamboos clean and understandable web interface. Consequently, Bamboo is chosen as the continuous integration tool for KIELER.

5.3.3 Installing Bamboo

Bamboo is installed to `/srv/Bamboo` by unpacking the distribution archive. Before it can be started for the first time, there is some configuration to be made on the system. For security reasons Bamboo will be run with the privileges of the *www-data* user. As Bamboo runs in a Java servlet container, the following environment variables have to be set for www-data in order to provide the proper Java version:

```
export JAVA_ROOT=/home/java/jdk64_1.6.0_21
export BAMBOO_USER=www-data
export JAVA_HOME=$JAVA_ROOT
export JAVA_BINDIR=$JAVA_ROOT/bin
```

The Bamboo web interface should be reachable under *http://rtsys.informatik.uni-kiel.de/bamboo*. As the webserver is running on another server, *mod_proxy* is configured to proxy requests on `/bamboo` and its subtree to the KIELER project management server. This is done in `/etc/apache2/mods-enabled/proxy.conf`:

```
ProxyPass /bamboo http://attosec:8085/bamboo
ProxyPassReverse /bamboo http://attosec:8085/bamboo
```

When proxying subdirectories on webservers it is best practise to proxy to the same subdirectory on the remote machine. In order to make Bamboo available in the `/bamboo` subdirectory on the project management server, the following settings are changed in `/srv/bamboo/conf/wrapper.conf`:

```
wrapper.app.parameter.4=/bamboo
```

This configuration parameter is rather cryptic and taken from the *Proxying* section of the Bamboo administrator's handbook [Atl12].

In order to store user and build data Bamboo needs a database. Bamboo ships with a built in database engine which serves for evaluation purposes. In a production environment,

Atlassian recommends to use either MySQL or Postgres, two popular database management systems. As on the KIELER server MySQL is already running to manage the source code repositories, a new database to store Bamboo data is set up by running the following commands on the MySQL shell:

```
create database bamboo character set utf8 collate utf8_bin;
grant all privileges on bamboo.* to 'bamboo'@'localhost' identified by 'secret';
```

The second SQL statement grants all privileges on the database *bamboo* to the database user *bamboo* that has to authenticate with a secret password.

This concludes the system configuration and Bamboo is started afterwards by executing `/srv/bamboo/bamboo.sh start` as the user www-data. After navigating to the KIELER Bamboo website, the last installation steps are taken directly in the webinterface. This includes setting database connection parameters, confirming the URL under which Bamboo may be reached, and configuring the local *LDAP* server as the authentication directory for Bamboo users.

5.3.4 Bamboo Build Configuration

In Bamboo, builds are categorised in different projects. Projects may contain multiple build plans, each consisting of one or more build stages executing the required build tasks. First of all a new project named *KIELER mainline* is created. Within this project multiple build plans are implemented:

Continuous Plugin builds The continuous plugin builds are triggered by each source code checkin. The build plan builds and tests all KIELER plugins. Three plugin builds are implemented, one each for Eclipse 3.7, 3.8 and 4.2

Nightly RCA builds The nightly builds run every night at a scheduled time to perform a full KIELER RCA build. Furthermore the KIELER P2 repository is built and both the repository and the RCA is deployed to the KIELER download site. As with plugin builds, three RCA builds for the different Eclipse versions are created.

Nightly QA build The nightly QA build runs static code checking on the KIELER sources to maintain code quality. It fails if a certain threshold of new Checkstyle or FindBugs warnings is exceeded.

Nightly KIELER rating build The nightly KIELER rating build runs a special Javadoc doclet to build the KIELER code rating website. Details about KIELER code rating will be given later.

Configuration of the build plans is done via the Bamboo web interface by clicking on the *Create Plan* button. After choosing a project and name for the new build plan, Bamboo's plan configuration interface is opened. On the configuration interface first the KIELER mainline Git repository is configured as the source of this plan. Afterwards, the build strategy is set to poll the repository for changes in intervals of 180 seconds. In the notification section it is configured to notify committers of failed builds and the first successful build after failed builds. Automatic branch detection is enabled so that developers can push their feature branches to see if they break the build before their integration in the master branch. Branches are configured to be removed after five days of inactivity. Then a new stage is added to the build containing the *compile and package* job.

The *compile and package* job consists of two tasks, first the current sources are checked out from the KIELER repository, and afterwards Maven is run in the `plugin` directory with `clean`

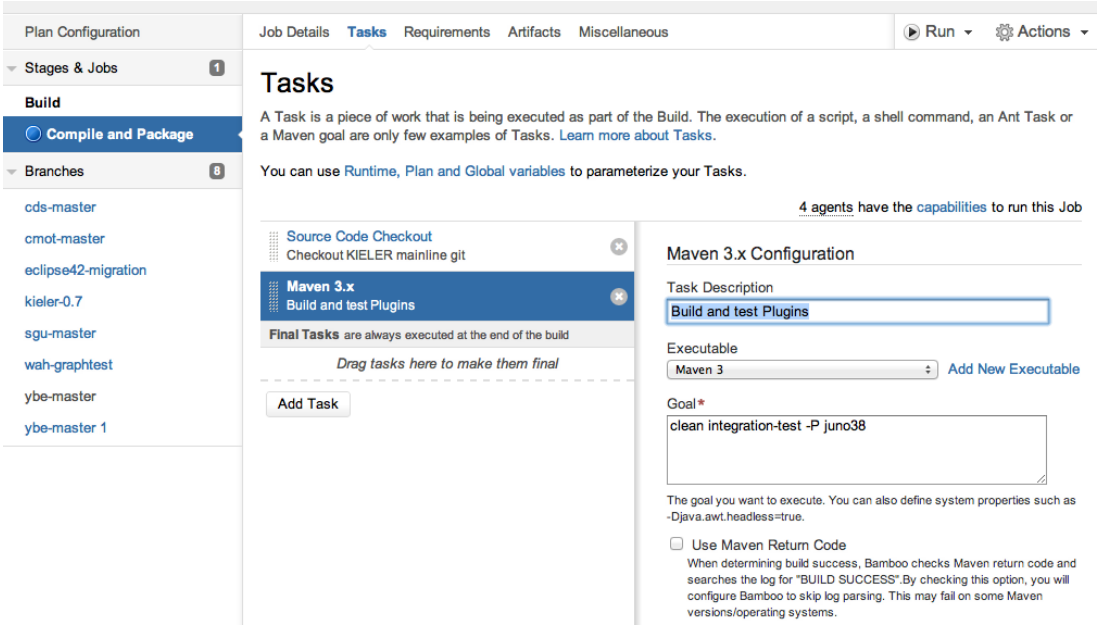


Figure 5.2. Bamboo task configuration

`integration-test -P indigo` as command line. This causes plugins to be built and tested. The `-P indigo` switch sets build profile to *indigo*, which is the Eclipse 3.7 build. Similarly two build plans for Eclipse 3.8 and 3.9 are created.

The KIELER RCA nightly builds are also created. Differently to the plugin builds, the build strategy is set to *scheduled* at 5AM and automatic branch detection is disabled. In addition to the build stage, the RCA nightly builds also contain a deploy stage. In this stage the KIELER RCA and repository are deployed on the KIELER download site via `rsync`.

The KIELER QA build and the KIELER nightly rating build differ mainly in the Maven command line from the RCA build. Both builds are run nightly on a fixed schedule. The KIELER rating website is also distributed with rsync. For the QA build the full Maven command line is `clean package pmd:pmd checkstyle:checkstyle findbugs:findbugs` and for the rating build it is `clean package javadoc:aggregate`.

Now it is possible for all developers to get an overview about the current build status on Bamboo's dashboard. Furthermore, developers committing changes that break the build process are notified so they can fix it promptly.

Chapter 6

Issue Tracking

Even the most careful developer's code sometimes contains software bugs. While the code might compile just fine, during run-time undesired behaviour may be observed. When working in larger development teams a system to track such bugs is required [JD03]. Ideally, such a system provides a web interface so that all developers can get an overview about current issues. Furthermore, an issue tracking system should be able to keep track of development tasks and planned improvements, and should also provide means to generate reports e. g., to keep track of how much work needs to be done until the next release, or about average ticket age. However, installing a proper issue tracking software is not enough. Even more important is the implementation of an efficient ticket workflow and to ensure that all developers make proper use of the system. If for example a developer fixes a bug and forgets to close the corresponding ticket afterwards, even the best ticket system will become cluttered and and disorganised. Therefore, it is very important to enforce proper usage of the issue tracking system by all developers. In this chapter we will first analyse the current situation and learn about problems of the current issue tracking process in the KIELER project. Afterwards, solutions and improvements will be discussed and suggested, both on the software side, as well as in the issue management process.

6.1 The Current Situation

Currently the KIELER project uses the *Trac* project management software in order to keep track of bugs. Trac is developed by Edgewall Software, released as open source software and licensed under a modified BSD style license. Trac is written in Python and runs on the group's webserver.

Each ticket in Trac can be identified by a unique ticket ID. Tickets are classified with distinct types, such as defect or enhancement, and contain also further information about for example the affected component or the target milestone. Currently there are 153 unresolved tickets of which 77 tickets are scheduled for the next release. There is a special milestone *The Indefinite Future* for issues that are recorded, but will possibly never be fixed. In commercial software development, having such a milestone would be inadvisable. However, as KIELER is an academic research project this is fine since there are not enough resources to resolve difficult issues that would require complex changes to the KIELER sources.

The interviews with the KIELER developers have shown that most developers tend to be unhappy with the functionality Trac provides. Developers complain about usability issues with Trac, for example writing queries to get a subset of open tickets is not user-friendly enough. When looking at the open tickets it becomes evident that the system is not properly used. There are for example tickets with comments like "This is fixed", which were nevertheless left open instead of properly closing them. Also, the average age of open tickets lies at about 150 days. The same holds for the average time from ticket creation until ticket resolution,

which also lies at about 150 days. This shows that in a new issue tracking system measures have to be taken to ensure that tickets are processed in a timely manner. If users of the ticket system get the feeling that creating a ticket does not cause any reaction, people will stop reporting new issues or even stop using KIELER. Some tests have shown that the reaction time on created tickets currently varies from two hours to no reaction in over one month.

6.2 Choosing a new Issue Tracking Software

As the interviews made clear developers are at best content with Trac as an issue tracker (see sec. 4.1). Therefore, as part of the effort of establishing a new issue management workflow, some alternatives have to be evaluated.

Redmine

Redmine is an open source, web-based project management and issue tracking tool, licensed under the terms of the GNU General Public License. It essentially provides the same set of features as Trac. There is a project wiki for documentation and a ticketing system, and Redmine can be connected to various source code management systems including Git, Subversion, and CVS. Furthermore, Redmine provides a calendar and *Gantt charts* to aid in visual representation of projects and milestones. Redmine is actively developed, runs on the *Ruby on Rails* framework, and supports multiple backend databases.

While evaluating Redmine for the KIELER project it has become evident that its features and even its user interface is very similar to Trac. As Trac has turned out to be unsuitable for KIELER development, chances are high that using Redmine would not provide any significant benefits. Therefore, switching to Redmine would not justify the expense of its installation and configuration.

Jira

Jira is a web-based issue tracking system developed by Atlassian software which is developed since 2002. Jira concentrates on issue tracking, contrary to Redmine and Trac it does not provide a wiki for project documentation. Jira is developed in Java and is shipped as a Java web archive running in a servlet container such as *Apache Tomcat*. Jira provides a clean web interface and issue workflow as well as ticket fields that can be freely defined. Furthermore, it provides configurable gadgets to visualise various ticket statistics such as average ticket age, created tickets vs. closed tickets, and other project statistics. These gadgets can be displayed on configurable *dashboard* pages. Jira can interoperate with other web-based tools from Atlassian, such as Bamboo and the web based source code repository browser Fisheye.

The evaluation of Jira has made clear that it provides many benefits compared to Trac. The configurable ticket workflow and fields allow fitting the issue tracker perfectly to the needs of KIELER development. The user interface is clean, efficient, and enjoyable to use. Through Jira's configurable dashboard pages it is possible for every developer to get an overview about the current development progress in various aspects. While being licensed under a proprietary commercial license, Atlassian provides free licensed for qualified open source projects. Therefore, Jira is chosen as the new project management and issue tracking platform for KIELER development.

6.3 Installing Jira

The Jira standalone distribution archive contains the Jira application together with all dependencies, such as servlet container and database drivers. After downloading, the archive is extracted to `/srv/atlassian-jira-5.0.4-standalone`. As with bamboo, Jira will be run in the context of the *www-data* user. The following Jira-specific environment variables have to be set for www-data:

```
export JIRA_USER=www-data
export JIRA_HOME=/var/atlassian/jira-data
```

This causes Jira to drop privileges after start and to use `/var/atlassian/jira-data` as the data directory.

By default, Jira runs on port 9080. In order to make the web interface available on the group's webserver under `http://rtsys.informatik.uni-kiel.de/jira`, the following lines are added to `proxy.conf` on the webserver:

```
ProxyPass /jira http://attosec:9080/jira
ProxyPassReverse /jira http://attosec:9080/jira
```

On the Jira server the context path for the Jira application has to be set accordingly in `conf/server.xml`:

```
<Context path="/jira" docBase="${catalina.home}/atlassian-jira"
      reloadable="false" useHttpOnly="true">
```

Jira stores all data in an SQL database and is shipped with an internal HSQL database. However, it is not advisable to run HSQL databases in production environments as they do not provide high performance, provide no comfortable means for backup, and had severe issues of data corruption in the past. As MySQL is already installed on the project management server , this installation will also be used for Jira as database backend. Therefore the Jira database is created on the MySQL shell:

```
create database jira character set utf8 collate utf8_bin;
grant all privileges on jira.* to 'jira'@'localhost' identified by 'secret';
```

Afterwards Jira is started by running `bin/start-jira.sh` from within the Jira installation directory. An installation wizard on the Jira web interface guides though the necessary post-installation steps, such as database settings, LDAP server settings for user authentication, and the Jira base URL.

6.4 Configuring Jira

Jira can serve as an issue tracker for multiple projects. Configuration in Jira is done per project. A project in Jira contains all project specific data, mainly tickets. In order to configure Jira for the newly created KIELER Jira project the following things have to be done:

- Choose a set of ticket fields that are to be displayed
- Set up a ticket workflow in Jira
- Prepare info screens containing statistical and other information about the project's current state

- Set up links to other Atlassian applications e. g., Bamboo
- Import old tickets from Trac

Ticket fields

When creating a new issue, the user is presented with a dialogue containing previously configured fields to enter detailed information about the reported issue.

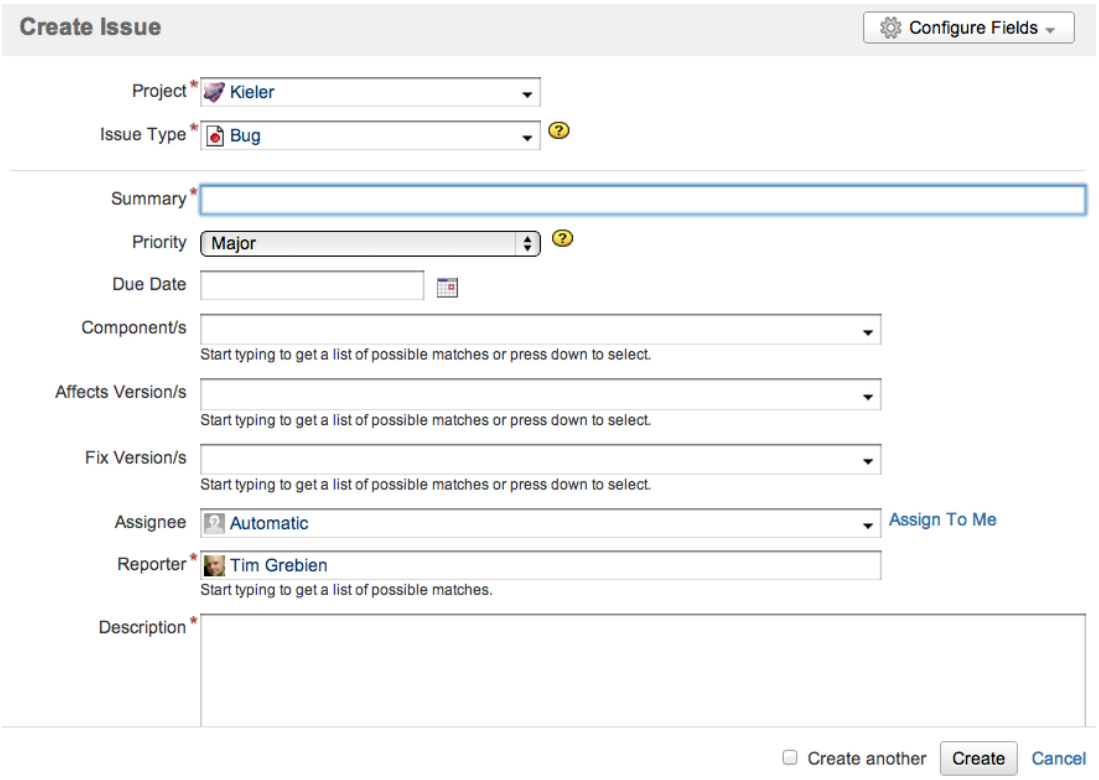


Figure 6.1. New ticket dialogue in Jira

Fields are configured in the *Field Configuration* section of the administration interface. Fields can be set to be mandatory or optional, depending on whether the field information is absolutely required to process the ticket. In the KIELER project the following mandatory fields are defined:

Issue Type The user may choose from a pre-defined list to specify the exact type of issue he is reporting. For KIELER, allowed values are *bug*, *new feature*, *task*, *improvement*, and *review task*.

Summary The user has to provide a short summary about the issue.

Description A more detailed description has to be provided. For bugs, this should for instance include the steps to reproduce [BJS+08].

Components Each issue has to be assigned to at least one component. This helps to get an easy overview about the number of issues in the different components of KIELER.

Assignee The assignee is responsible for the issues assigned to him or her. He or she can either resolve the issue or re-assign it to another person.

Severity Issues need to be assigned a severity. This allows classifying issues from *minor* to *blocker*. Blockers for example are issues that prevent further development on certain components or even on the project as whole.

Furthermore, there are optional fields for issue linking, fix version, due date etc.

Ticket workflow

Jira allows the definition of a ticket workflow. Each issue in the Jira system has a certain status assigned to it. Ticket workflow in Jira means defining how status changes from issue creation to issue resolution may be performed. The following image shows the KIELER ticket workflow as implemented in the *Jira Workflow Designer*.

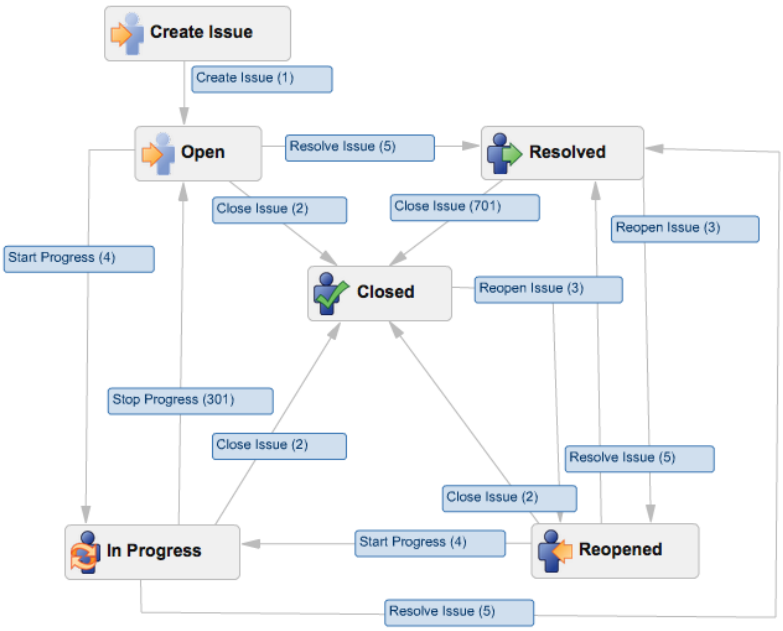


Figure 6.2. The KIELER ticket workflow

A typical issue lifecycle in KIELER could look like this. After creation, the issue is first assigned the status *open*. As soon as the assignee starts working on the issue he may change its status to *in progress*. This is mostly done if work on the issue will take some time and the assignee wants to indicate that the issue is being taken care of. If the issue can be resolved directly, the assignee can either close or resolve the issue. Both states, *closed* and *resolved* indicate that work on the issue is finished and the issue is resolved. The reason for having a special status *resolved* is that if the assignee might want the reporter or some other person to take a look to confirm that the implemented solution is sufficient. When resolving or closing a ticket, the assignee has to choose a *resolution type*, this might for example be *fixed* if the issue was fixed, or *cannot reproduce* if he was unable to reproduce the issue.

Importing from Trac

Jira provides a plugin to import issues from other issue trackers. To import issues from Trac, the *Trac Environment* which contains the database and configuration files for a certain Trac project has to be compressed and uploaded to Jira; in case of KIELER this means compressing and uploading the directory `/srv/trac/kieler`. Afterwards the *Jira Trac Import Wizard* guides through the necessary steps. First the old KIELER Trac project has to be mapped to the KIELER Jira project. Afterwards Trac issue fields are mapped to the corresponding issue fields in Jira. As issue linking in Jira works differently from Trac, the fields *Blocked by* and *Blocking* are first imported as custom fields and afterwards issues are linked manually in Jira. As there are not many issues affected, this step has been less tedious than first thought.

6.5 Social Aspects

The first parts of this chapter described how to tackle the technical difficulties of issue tracking and the implementation of a software environment to help implement it. However, the best issue tracker implementation will not help when used improperly. The interviews with the developers have shown the need for keeping the issue tracker tidy. A good technical basis with a clear web interface helps the new issue tracker to be accepted by the developers [BVGW10]. Additionally, further measures are introduced.

First of all a person is appointed the role of a project secretary, as past experience has shown the need for a person to oversee the project guidelines. The project secretary should not be chosen from the group of developers as he should keep an impartial overview about the whole ticket situation, and not just over the tickets that are assigned to him as it is usual for developers. The secretary is responsible for ensuring that tickets are processed in a timely manner or at least postponed to the *Indefinite Future* milestone. Furthermore he has to ensure that the implemented ticket workflow is correctly adhered to. In the weekly KIELER meetings he gives an overview about the current project's state regarding open tickets.

To prevent tickets from being unprocessed or unnoticed for a long time, the usage of the *due date* field is enforced. New issues should be created with a due date of one week, which implies that the assignee should react to a new issue within one week. Reaction in this case does not necessarily mean that the issue has to be resolved within on week, confirmation of the issue by leaving a comment would for example be sufficient. The project secretary regularly filters the open tickets for overdue tickets and contacts the corresponding developers.

Another measure to prevent cluttering of the issue tracker is to monitor the special indefinite future milestone closely. The project secretary should schedule regular meetings every three months to go through all tickets which are scheduled for the indefinite future milestone. For each ticket it should be decided if it has become obsolete, if it can be rescheduled for the next release, or if it is left as is. This prevents issues from being completely forgotten and hence the accumulation of too many tickets in the special future milestone.

Chapter 7

KIELER Documentation

A common problem, especially in academic software projects, is to maintain documentation that is both comprehensive and up to date. Most of KIELER's code is written in the context of student or PhD theses. Writing documentation can thus become tedious as all developers have many other things to attend to. The interviews have shown that most developers think that the KIELER documentation should be improved. Currently the documentation is kept in the KIELER Trac project wiki. As issue tracking in KIELER has been moved from Trac to Jira, it is needed to implement a new software solution for managing the documentation. As many people are working on KIELER and the documentation should be available in form of a website, keeping the documentation in a wiki is the most appropriate solution.

Atlassian provides *Confluence*[1], a feature-rich wiki software which is especially suited for keeping documentation. As the other Atlassian tools, Confluence is written in Java and runs in a servlet container. Contrary to other wiki software, Confluence does not support a wiki markup language edited directly in a web-based pure text editor. Wiki pages are instead edited in a rich text editor supporting special markup commands which are transparently translated and shown in the editor. If for example the user types `.h2`, the following text is automatically converted to a second-level headline.

This chapter first describes the installation and configuration of Confluence. Afterwards, it explains how the KIELER documentation is migrated and restructured. Finally, methods of keeping the documentation up to date are investigated.

7.1 Installing Confluence

The Confluence standalone installation package is downloaded and afterwards extracted to `/srv/atlassian-confluence-4.2.2`. The package contains the Confluence application packed as a *Java Web Archive* together with the Apache Tomcat Java servlet container. After unpacking, the Confluence data directory is set in `WEB-INF/classes/confluence-init.properties`:

```
confluence.home=/var/atlassian/confluence-data
```

By default, Confluence runs on Port 8090. In order to make confluence available in the `/confluence`-subtree on the Rtsys Webserver, the context path is set in `conf/server.xml`:

```
<Context path="/confluence" docBase="../confluence" debug="0"
      reloadable="false" useHttpOnly="true">
```

Furthermore proxy settings have to be made in `proxy.conf` on the Rtsys webserver:

```
ProxyPass /confluence http://attosec:8090/confluence
ProxyPassReverse /confluence http://attosec:8090/confluence
```

[1] `http://www.atlassian.com`

Confluence uses MySQL as backend database engine; therefore the database and a corresponding user are created on the MySQL server:

```
create database confluence character set utf8 collate utf8_bin;
grant all privileges on confluence.* to 'conflu'@'localhost' identified by 'secret';
```

Afterwards Confluence is started by running `bin/start-confluence.sh` as user `www-data` and the installation is finalised on the Confluence web interface.

7.2 Configuring Confluence

Confluence is capable of hosting wikis for multiple projects in so called *wiki spaces*. Therefore, the first configuration step is to create a new wiki space *KIELER* to host the KIELER documentation. Confluence provides pre-defined page layouts suitable for different purposes. As the KIELER wiki space hosts documentation, the *Documentation Theme* is selected.

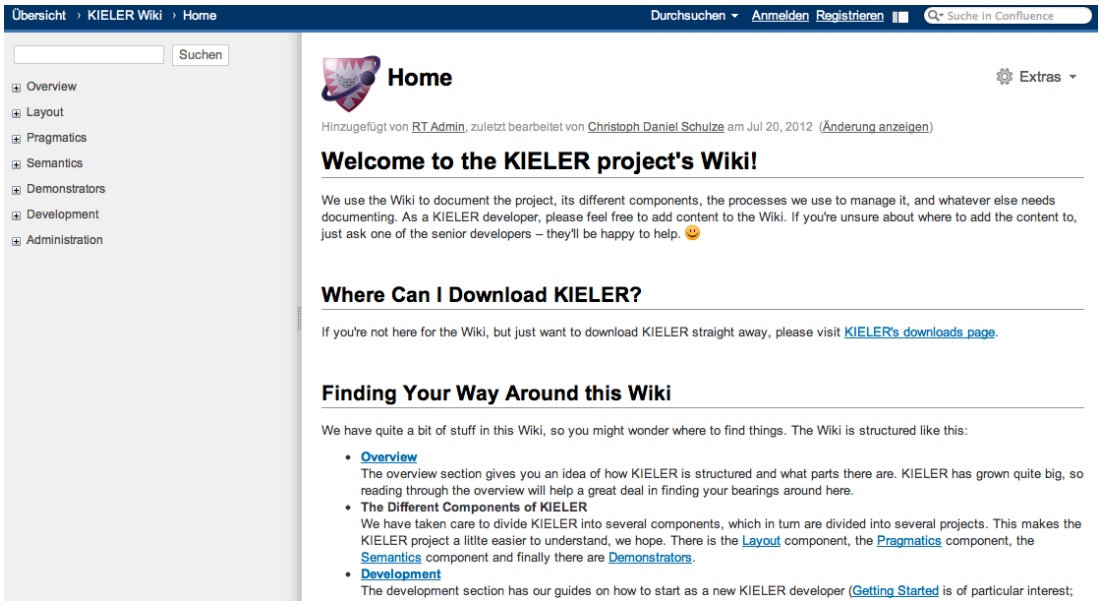
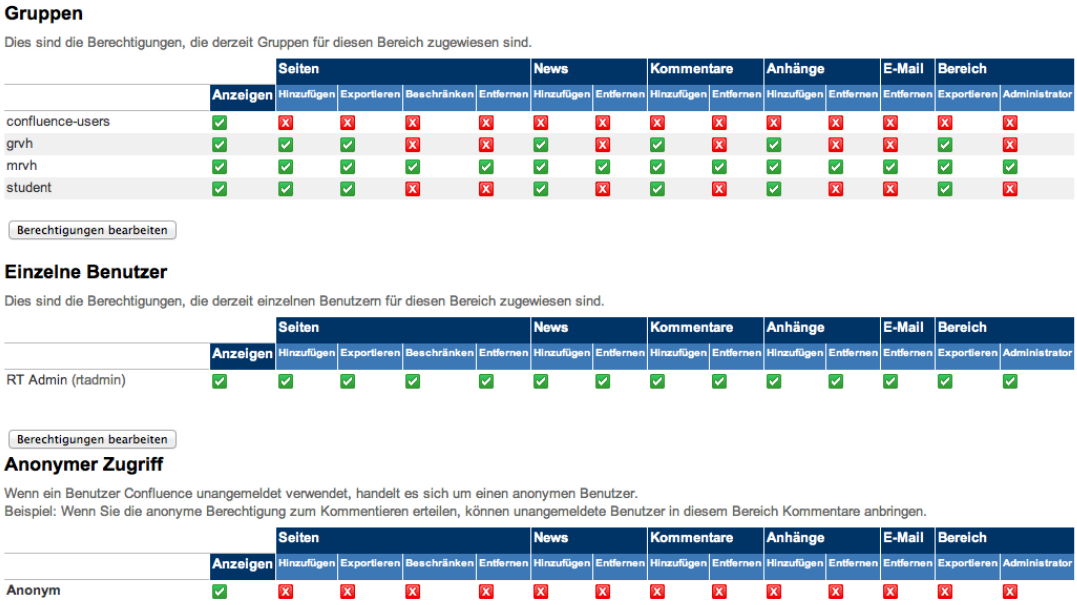


Figure 7.1. The KIELER documentation wiki

The documentation theme provides a two-column layout: in the left column a page tree is displayed to allow quick navigation in the wiki, while in the right column the actual wiki page is presented. To allow inclusion of information from Jira and Bamboo, application links to Jira and Bamboo are created on the *Confluence Space Admin* pages.

By default Confluence wikis are writable by anyone, including unauthenticated users. To prevent unauthorised editing, certain permissions for existing LDAP groups are set in the permission interface. Unauthenticated users are restricted to the *View* permission and hence may not modify any content in the KIELER documentation wiki.



Gruppen

Dies sind die Berechtigungen, die derzeit Gruppen für diesen Bereich zugewiesen sind.

		Seiten				News		Kommentare		Anhänge		E-Mail	Bereich	
	Anzeigen	Hinzufügen	Exportieren	Beschränken	Entfernen	Hinzufügen	Entfernen	Hinzufügen	Entfernen	Hinzufügen	Entfernen	Entfernen	Exportieren	Administrator
confluence-users	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
grvh	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	✗	✓	✗
mrvh	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
student	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	✗	✓	✗

Berechtigungen bearbeiten

Einzelne Benutzer

Dies sind die Berechtigungen, die derzeit einzelnen Benutzern für diesen Bereich zugewiesen sind.

		Seiten				News		Kommentare		Anhänge		E-Mail	Bereich	
	Anzeigen	Hinzufügen	Exportieren	Beschränken	Entfernen	Hinzufügen	Entfernen	Hinzufügen	Entfernen	Hinzufügen	Entfernen	Entfernen	Exportieren	Administrator
RT Admin (rtadmin)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Berechtigungen bearbeiten

Anonymer Zugriff

Wenn ein Benutzer Confluence unangemeldet verwendet, handelt es sich um einen anonymen Benutzer.
Beispiel: Wenn Sie die anonyme Berechtigung zum Kommentieren erteilen, können unangemeldete Benutzer in diesem Bereich Kommentare anbringen.

		Seiten				News		Kommentare		Anhänge		E-Mail	Bereich	
	Anzeigen	Hinzufügen	Exportieren	Beschränken	Entfernen	Hinzufügen	Entfernen	Hinzufügen	Entfernen	Hinzufügen	Entfernen	Entfernen	Exportieren	Administrator
Anonym	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Figure 7.2. The KIELER wiki permission scheme

7.3 Migrating and Restructuring the KIELER Documentation

Once Confluence is configured properly, the KIELER documentation needs to be migrated from the old Trac wiki to the new Confluence wiki. Unfortunately Confluence's wiki import plugin does not work properly for Trac wikis. Tests have shown that especially the table import does not work without errors. As there are many tables in the KIELER Trac wiki, migration has to be done manually. However, this is not a big drawback since a manual migration gives the opportunity to rework all wiki pages during the migration step. This also allows restructuring the documentation.

In the interviews the developers stated that the Trac wiki structure needs improvement especially in terms of structure to help new developers to find the right pages. Therefore, a new structure for the KIELER documentation is developed first.

7.3.1 Trac Wiki Structure

The first approach to getting a feeling of how KIELER documentation is structured in Trac is by looking on the Trac wiki front page. In the page's upper right corner there is a box providing quick navigation to non-documentation pages such as downloads, pointers to the KIELER mailing list, etc. The documentation pages for KIELER's sub-projects are listed in a table which classifies documentation pages into three sections: *Semantics*, *Syntax*, and *Pragmatics*. Besides listing the sub-project's acronym and full name, there is no further description of what a sub-project actually does. Beneath the table there is a section called administration providing links to the developer start page and a link to a page that describes acronyms used in KIELER. Navigating to the distinct project subpages shows that documentation is rather

brief and could be improved.

The other approach is to look at the special *TitleIndex* page that contains a tree of all pages in the corresponding Trac wiki. Looking at the TitleIndex makes clear that the KIELER documentation tree is currently structured like this:

Help Contains mostly how-tos explaining how certain things can be done on the Eclipse platform with the third party plugins that KIELER depends on. It also contains a software practise section, which contains the KIELER coding and project guidelines.

Projects Contains information on all KIELER sub-projects. The projects themselves are not structured further within the subtree.

Releases Contains release notes for the past KIELER releases.

Meetings Contains protocols of the weekly KIELER meetings and protocols of review meetings. There is, however, also a distinct section for reviews which means there are inconsistencies within the structure.

Besides this structure there are also some top level pages that are not properly categorised or do not fit into the categories described above.

7.3.2 The new KIELER Documentation Structure

When designing a structure for documentation wikis, the most important thing is to think about how the documentation is structured. It is very important that especially new developers are able to find their way around the documentation easily. Keeping this in mind, the wiki's start page will contain an overview over the basic wiki structure which will be described in the following sections.

Overview

The overview section should give an idea about what KIELER does and about the internal structure of KIELER. Therein, it is described how the different components of KIELER work together and depend on each other. This section will be the smallest section containing only a top-level page, an about page, and the KIELER release notes as sub pages.

Components

This part of the new KIELER documentation wiki will contain the documentation for all KIELER components. Discussions with the lead developers have shown that all subprojects can be sorted into four subcategories.

Layout Contains the KIELER infrastructure for automatic graph layout, graph analysis and the KIELER layout algorithms

Pragmatics Contains components that reflect the KIELER research effort on modelling pragmatics. This comprises creation and modification as well as the synthesis of different views on models.

Semantics Contains the KIELER infrastructure to define execution semantics for meta models. Also contains simulators for *Esterel*, *S*, *UML* and some more languages

Demonstrators The KIELER demonstrators are mainly editors to demonstrate the technologies developed in the first three sections, most notably *ThinKCharts*, the KIELER SyncCharts editor.

All of KIELER's subprojects will be sorted into the above sections. In order to achieve a common appearance, all project pages have to include a common project information header that names the responsible developer and lists theses that are related to a given project.

Administration

The administration section will contain documentation on all administrative aspects in KIELER. This includes descriptions of the KIELER software development process including build management, the release process, source code management, etc.

Development

The development section will document the application of the processes defined in the *Administration* section above. It will contain a subsection for project guidelines, including the KIELER coding and naming conventions, information on configuring Eclipse, and information on using Git. Meeting protocols will also be kept in this section.

7.3.3 The Migration Step

Now that a new documentation structure is designed, the real migration step can take place. As Confluence supports no automatic import of Trac wiki pages, all pages will have to be migrated manually from Trac. Fortunately, tests have shown that when using copy and paste, most page formatting such as headlines and font face are also copied. As the first migration step, dummy pages containing only an error box that states that the page has not been migrated yet are created for each KIELER project. Afterwards, all developers that are responsible for a KIELER subproject or component are contacted and told to migrate and revise their parts of documentation. As there is no *Administration* and no *Development* section in the KIELER Trac wiki, all pages there have to be written mostly from scratch or with information found throughout the old Trac wiki.

7.4 Keeping Documentation Up-to-Date

After the successful migration of the KIELER documentation from Trac to Confluence, certain measures have to be taken to prevent documentation from becoming outdated. In the issue tracking section a project secretary for KIELER has been introduced. This secretary also has to ensure that documentation is maintained. He should schedule regular documentation reviews in which the project responsibles are prompted to go through their documentation. These regular reviews will take place during the release process and will be detailed in the corresponding section.

Furthermore, the project secretary will monitor development for new features and components to ensure that new parts in KIELER are also documented in the wiki as part of the development process. This includes reminding students that writing documentation is also part of their thesis.

Chapter 8

Code Quality in KIELER

Maintaining high code quality in large academic software projects is a very important but also very tedious task. Development in such projects is mostly done as part of research and also student theses. For example, a bachelor thesis is written over the course of half a year, which means that many developers are part of the team only for a very short period. Furthermore, time is a precious resource when writing a thesis, and often source code refactorings, writing unit tests, and writing documentation within the code is left out in favour of other things. Additionally, being a research project means that development includes trying out things that might not even work properly. Also, work on the code is mostly done by individuals. There is no collective code ownership, so it is quite possible that there are pieces of code in KIELER that nobody besides the developer who wrote the code has seen yet.

All these aspects make it even more important to monitor code quality closely. However, as the members of the research staff all have many responsibilities besides KIELER development and are also working on KIELER themselves and are therefore biased, the appointed project secretary has to take responsibility for maintaining code quality in KIELER. Most importantly, this includes monitoring that the KIELER project guidelines are followed, scheduling code reviews, encouraging developers to do design reviews, and the application of static code checkers on the KIELER code.

8.1 Code Reviews and KIELER Code Rating

A code review is the systematic examination of source code, conducted to find possible defects in the code and also to improve the overall code quality by ensuring that code is properly commented and development guidelines are adhered to. Code reviews can be done in various ways that may be categorised as formal code reviews, pair programming, and lightweight code reviews [HK07].

Pair programming is a technique that is commonly used in extreme programming, an agile software development technique. While one developer, the *driver*, writes the code, a second developer observes and reviews each line of code as it is written [BA04]. Roles are then switched frequently. For KIELER, pair programming is an unsuitable technique as code is written individually due to the fact that much development in KIELER is done as part of some thesis and therefore individual work.

Formal code reviews are based on a detailed and formal process with multiple participants that take certain roles in the review process. Often such a formal code review involves multiple phases with multiple meetings. A popular formal process is the *Fagan Inspection* [Fag76]. In a Fagan Inspection, the following roles are fulfilled by members of the development Team:

Author The person who has written the code.

Reader The reader presents the code to the audience.

Reviewers Multiple reviewers review the code line by line.

Moderator Observes the review session, cuts down lengthy discussions, and writes a protocol.

The review process then undergoes multiple phases. First of all, the author is responsible for planning and preparing the review, including choosing code to review and finding team members to participate. In the overview phase, the author introduces the other participants to the review objectives and to the code to be reviewed. Afterwards, in the preparation phase all participants prepare the review, by reading and understanding the code. In the following inspection meeting, the code is reviewed line by line and defects are marked. Afterwards the author reworks his code and in the followup phase the moderator verifies that all defects are fixed.

Lightweight code reviews use a less formal approach and are often conducted as part of the development process. Lightweight reviews can be done as *over-the-shoulder* reviews, with the author presenting the code to a fellow developer that literally looks over the author's shoulder. Often lightweight reviews are tool assisted: there are special tools for doing online reviews or for mailing around new source code after each checkin to the source code management system. A case study by Cohen has shown that lightweight review processes can uncover as many bugs as a formal process while being more cost-efficient and faster.

The *Ptolemy* project has shown that the application of a formal review process on academic software projects is possible [RNHL99]. Learning from Ptolemy, an effort has been made in 2009 to adapt the Ptolemy review process to KIELER. Unfortunately this attempt has failed. From the protocols in the old Trac wiki it could be gathered that only about ten code reviews took place from 2009 until 2010, while in 2011 only design reviews were conducted. Code reviews took about three hours to complete and often enough five to six developers were involved. From the interviews with the developers the following reasons that caused the process to fail have been gathered:

- The authors were responsible for initiating code reviews, which in practise only rarely happened.
- Reviews tended to take too much time, due to lengthy discussions during the review sessions.
- Reviews were badly prepared as authors did not know enough about how the complicated process works.
- Roles were not fulfilled properly.

As the formal process has been found to be too difficult and too time-consuming for KIELER, the new review process will be a lightweight process. Furthermore a code rating is introduced to keep track of the current review coverage and code stability.

8.1.1 KIELER Ratings

Inspired by the Ptolemy project a colour rating for classes is introduced [RNHL99]. The colours serve as indicators for code maturity as well as they reflect a class's code review state. The following rating scheme will be used for KIELER:

Red Red classes have never been reviewed and are possibly not stable regarding design and implementation. All classes are red at first.

Yellow Yellow classes have undergone at least one review. While changes to design and API may occur, developers should be careful. When doing larger changes to yellow classes, the corresponding developer is responsible for conducting a short code review in order to ensure code stability. This is the desired colour for all classes in KIELER.

Green Green classes are considered stable. Multiple reviews must have already taken place and another thorough review is needed to rate a class as green. Green classes must not be changed API-wise, unless discussed in the weekly KIELER meeting. If changes to green classes occur, all changes have to be reviewed thoroughly. Also there have to be regression tests for green classes that run with each automatic build.

Blue Blue is the highest rating a class can reach. A blue rating is only awarded in special cases where a class is considered absolutely complete. No changes whatsoever should be made to blue classes. Furthermore, a blue class must meet the same requirements as green classes.

In order to keep track of class ratings a special Javadoc tag `kieler.rating` is introduced. The syntax of the tag is as following:

```
@kieler.rating <colour> <date> <review id> <reviewers>
```

This makes the generation of a rating overview website through a custom Javadoc doclet possible. The publicly available website should also encourage authors to conduct reviews in order to improve their class ratings.

8.1.2 KIELER Code Reviews

As discussed before, conducting code reviews helps to maintain high code quality by eliminating latent defects and violations of the coding rules. We have also learnt that a review process for KIELER has to be lightweight in order to be accepted by the developers. In the interviews all developers stated that they even see a personal gain in review sessions, both as a reviewer and also as the author. The new KIELER review process will be based on the following main ideas:

- Tool supported reviews to minimise preparation times for authors and reviewers
- Regular weekly review sessions, with a fixed meeting spot and time constraints on review duration

Regarding the second item it has been decided that Meetings will take place on Thursdays at 10 a.m. and will take at maximum 1.5 hours. Nine days before the review both author and two reviewers are appointed. The author chooses at most four vital classes containing about 300 to 400 lines of vital code. Vital in this case means functional algorithmic code without for example imports, class definitions, constant definitions, comments, etc. This allows the reviewers to spend about 1 to 1.5 hours on the review. Reviewers should complete their reviews two days before the review meeting takes place for the author to have time to study all the comments.

In the review meetings, the review moderator reads all comments to both reviewers and authors and leaves time for both to discuss questions that might have been arisen from the comments. Also the moderator ensures that review meetings do not lead to lengthy discussions and creates Jira issues for identified defects.

Introducing Crucible

Crucible is a web-based collaborative code review tool developed by Atlassian[1]. As the other Atlassian tools it is written in Java and runs in a Java servlet container. Crucible is part of the Fisheye tool, a repository browser for various source code management systems. Both Fisheye and Crucible are installed on the KIELER project management server and can be reached under `http://rtsys.informatik.uni-kiel.de/rtsys/fisheye`.

Reviewing source code with Crucible can be done in various means. Crucible supports traditional reviews where full source code files are put together in one Crucible review. Invited reviewers then read the code directly in crucible and can leave comments for each line of code. Comments can be flagged as a defect together with a classification of the defect, such as for example *risk-prone* or *not conforming to standards*.

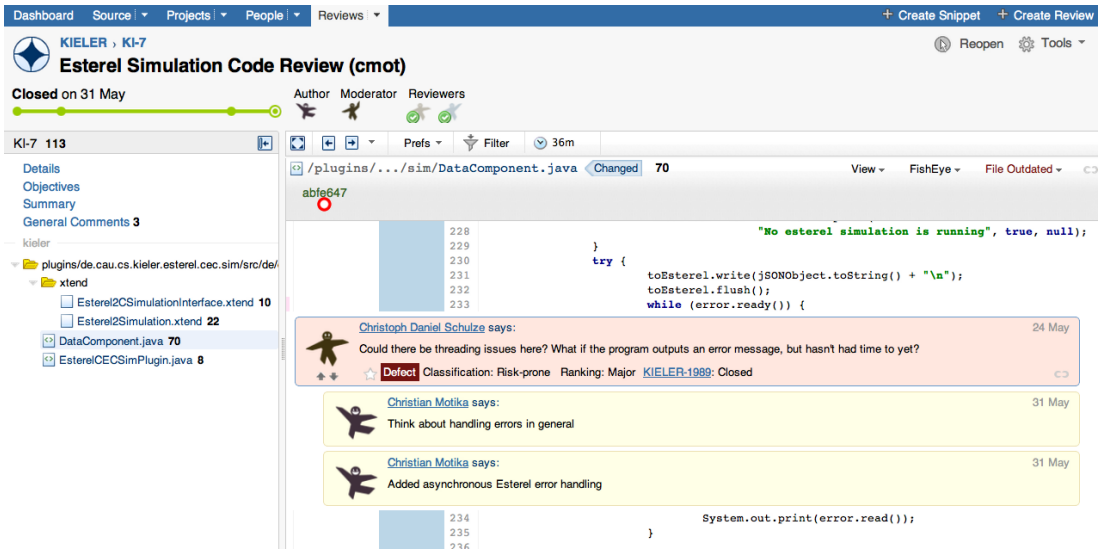


Figure 8.1. A code review in Crucible

Also the author or other reviewers may reply directly to a comment so that discussions can take place directly in Crucible. Reviewers can create new Jira issues directly from a comment so that tasks that arise from a Crucible review are tracked in Jira. When a reviewer has finished reviewing, he marks himself as complete. After all reviewers have finished reviewing, the author goes through the comments, repairs all defects, and writes a short summary before closing the review.

Instead of reviewing whole files, Crucible can also be used to review only differences between file revisions. This is especially helpful when conducting follow-up reviews or when re-reviewing already reviewed classes. It is also possible to generate reviews directly from Jira issues and Bamboo builds to review issue resolution and source code changes in single builds.

[1] `http://www.atlassian.com/`

KIELER reviews with Crucible

Putting it all together, the KIELER review process with Crucible is defined as follows. After appointing both the author and the two reviewers during the weekly KIELER meeting, the author first creates a special *Review Task* issue in Jira. Afterwards, he creates a review in Crucible and adds the files to be reviewed. He then links the review to the corresponding Jira issue and invites the formerly appointed reviewers. The review moderator is always the project secretary.

Then the reviewers read through the sources and leave comments in Crucible. A comment is marked as *defect* when the reviewer judges that the corresponding code has to be changed. This is no indicator for the severity of the issue: a defect might well just be a missing or too brief comment which is unacceptable when striving for high code quality. During the review week the author may also reply to comments if any questions have arisen. This also leads to a shorter review meeting as most questions may already have been answered during the review period. After reviewers have finished their reviews they mark themselves as complete in Crucible.

When all reviewers are finished, the author takes time to go though the comments and prepare for the review meeting. In the actual meeting the moderator reads all comments to the audience and leaves space for discussion.

Afterwards, the author documents his decisions by replying to each comment in Crucible and resolving the corresponding Jira issues. Then the author closes the corresponding review task in Jira and writes a summary in Crucible before closing the review.

Outlook on code reviews

The review scheme as defined above is designed to be as lean as possible. The idea is to establish a good and prosper review culture in KIELER by making reviews less time-consuming and most effective. At some point in the future, when all developers are used to the review scheme and reviewing code is part of the common development process, it might be possible to do code reviews purely online, without a meeting. Also authors should be encouraged to conduct mini-reviews on their own, where only changesets are reviewed to raise their personal code quality.

8.2 Design Reviews

Design reviews are held to verify and discuss a certain software design. As large parts of KIELER are developed as part of theses, it can be expected that developers are especially interested in good software design. Software design also often directly affects functionality, especially when implementing graphical user interfaces. Students and also the research staff are expected to discuss their designs with their advisers and fellow developers. Therefore, no formal process will be described in this thesis.

However, it is advised that each student thesis should undergo at least one design review. The results of this review should be recorded on a Confluence wiki page. This also helps to document certain design decisions, and therefore leads to a more comprehensible design.

8.3 Static Code Analysis

Static code analysis describes the process of analysing source code through software tools that looks for possible defects by for example searching for known bug patterns or for violations of coding standards [Lou06]. Contrary to dynamic code analysis, the code is not executed but only analysed. Static code analysis can also mean to apply formal methods on software, such as *model checking* or *dataflow analysis* for formal verification of certain safety-critical properties.

In KIELER, static code analysis will only be used to check if coding standards are adhered to and to identify possible bugs. All KIELER developers are advised to run both *Checkstyle* and *FindBugs* regularly on their code.

FindBugs

FindBugs[2] is an open source Java tool developed at the University of Maryland that searches for possible bug patterns in Java code. FindBugs is actively developed, and is currently at version 2.0.1. FindBugs can find hundreds of different bug patterns and can also identify inefficient code. Found defects are classified by severity and sorted into distinct categories, such as *Bad practice* or *Performance*. FindBugs also provides an Eclipse plugin for convenient usage from within the IDE. FindBugs operates on Java bytecode: sources are first compiled and then analysed. Afterwards the user is presented with the results.

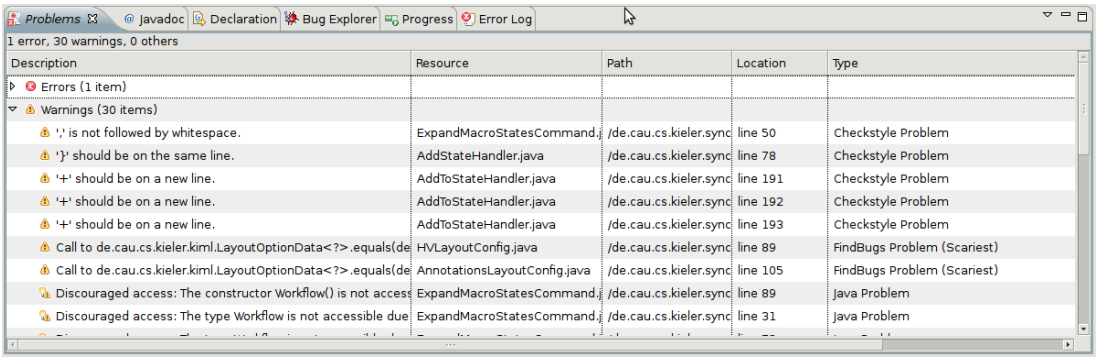


Figure 8.2. FindBugs and Checkstyle warnings in Eclipse

Checkstyle

Checkstyle[3] is an open source Java tool designed to check if Java source code complies with coding rules. Checkstyle is actively developed, and is currently at version 5.5. Checkstyle operates directly on the Java source code. It is configured through an XML configuration file named `checks.xml`. There is an Eclipse plugin that applies CheckStyle transparently on Eclipse projects without the need of explicitly calling it. Warnings are presented to the developer in the Eclipse problems view.

[2] `http://findbugs.sf.net`
[3] `http://checkstyle.sf.net`

8.3.1 Application to KIELER

As stated before, all KIELER developers are advised to run both Checkstyle and FindBugs on their code. However, the manual application of both tools on all of KIELER's plugins has revealed that over 90 of 150 plugins contained Checkstyle and/or FindBugs warnings. To change that one Jira ticket was created for each plugin with warnings to inform the responsible developers. During the course of resolving these issues it became evident that as FindBugs only searches for *possible* bugs, not all FindBugs warnings needed fixing. However, as FindBugs operates on Java bytecode, it is impossible to suppress the warnings by a simple comment in the corresponding line of code, because comments are not retained in the bytecode. Therefore, it has been decided that a clarifying comment has to be left in the code for every warning that is to be ignored.

Contrary to FindBugs, Checkstyle only searches for violations of the coding rules. There are only rare occasions in which fixing a Checkstyle warning does not make sense. Therefore, nearly all Checkstyle warnings have been fixed. If a Checkstyle warning remains unfixed, a special comment is left in the source code that suppresses the warning.

Learning from this experience it becomes clear that usage of static code analysis has to be monitored closely by the project secretary. He should run both tools regularly on all of KIELER's plugins to ensure that all coding rules are adhered to. Also, he runs both tools on code that is to be reviewed.

Furthermore, it would be advisable to find a solution for integrating code analysis in the nightly build process properly. While this has been done already, KIELER contains many lines of generated code, which produces many warnings in both Checkstyle and FindBugs. Therefore, nightly code analysis currently only serves as an indicator to see if the combined number of warnings increases or decreases.

Chapter 9

Managing Sources and Conducting Releases

Releasing software is a crucial part of the software development process. A software release marks a certain stable milestone in the software development cycle. In KIELER, the release process has always been a weak spot. Release dates that had been set always had to be postponed for at least a month. The interviews with the developers indicated that this was mostly due to bugs that appeared during the pre-release testing process, which should have been already found earlier. Also certain features that were not ready yet were scheduled for release.

A proper and lightweight release process is closely tied to source code management. When the main development line is always in a mostly stable state, theoretically a release can occur at any time. Therefore the first part of this chapter deals with how sources should be managed in Git before describing a release process for KIELER.

9.1 Source Code Management in Git

As we have learnt before, the KIELER sources are managed with the distributed version control system Git[1]. Git is an open source program that was initially designed and developed for Linux kernel development. As a distributed system, Git does not depend on a central server or network access: each Git working copy contains the current state of the managed source code, but also its complete history and revision log. One of Git's strengths is the strong support for branching and merging. Creating a new branch opens a new line of development starting from the repository's current state.

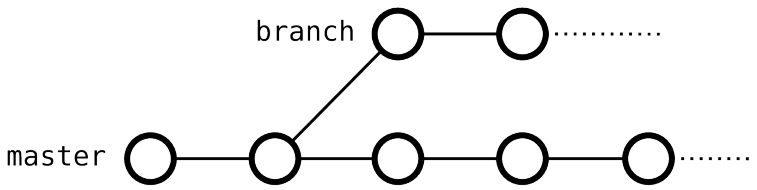


Figure 9.1. A branch in Git

Therefore, when working on two different branches, development happens completely independently until the branches are merged back together. While developing on a branch it

[1] http://git-scm.com

is also possible to continuously merge upstream changes back into the development branch to ease later merging. By default, every Git repository contains a branch called *master* which is the authoritative branch, and other development branches are typically forked from the master branch [Loe09].

For KIELER, a central authoritative Git repository is hosted on the KIELER project management server. The typical workflow of a developer is to first *commit* his source code changes into his local repository before *pushing* them to the authoritative repository on the server. Until now the minimum requirements for newly committed source code were that it must not break the build. However, as Git provides strong support for branches, this feature will be utilised in future KIELER development. The new KIELER source code management process aims at always having a having a master branch which can be considered release-ready. This will be done by shifting the main KIELER development away from the master branch to feature branches. The following classes of branches are defined:

- The master branch
- Release branches
- The production branch
- Hotfix branches
- Feature branches

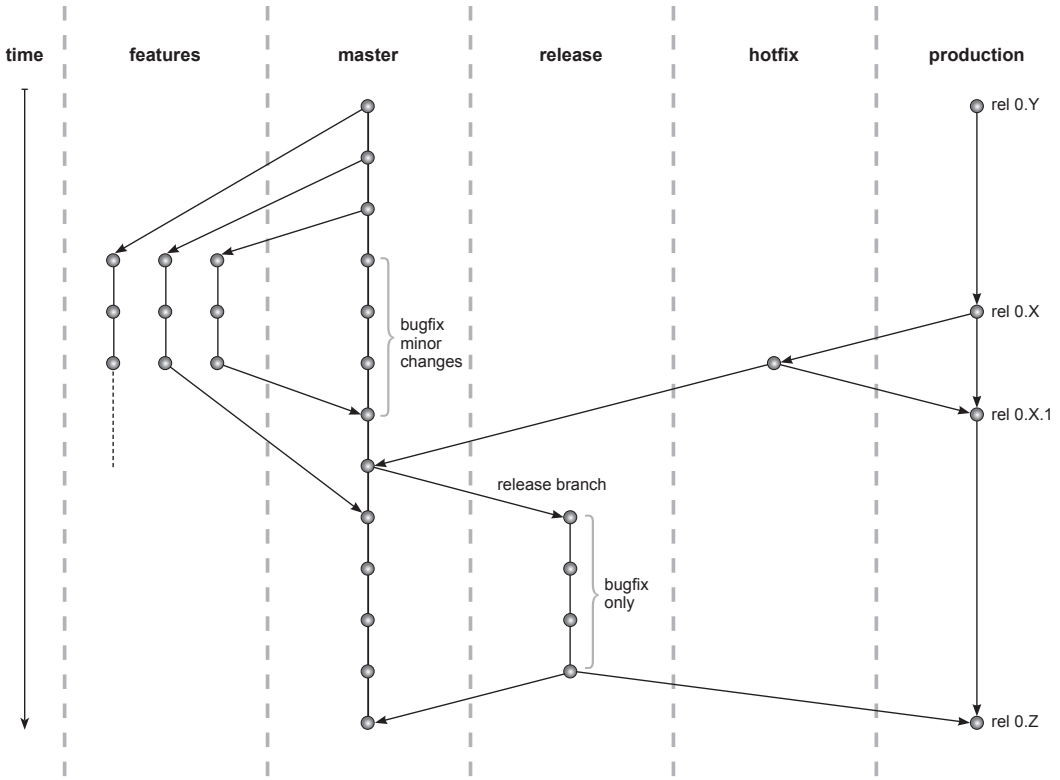


Figure 9.2. KIELER branching scheme

The master branch

The master branch is the authoritative branch for KIELER development. On the master branch only bugfixes and minor features are pushed directly. The master branch must always be in a release-ready state. This means that all features that are merged into the master branch should be stable and complete. Feature branches are forked by the developers from the master branch. Also, if a release is planned, a release branch is forked from the master branch prior to the release.

Release branches

Prior to each release a release branch is forked from the master branch. No development of features takes place there, only bugfixes are pushed to the release branch. After a release is made, the release branch is merged both into the production branch and into the master branch. Afterwards, the release branch is deleted.

The production branch

The production branch's head is always the code of the most recent release. Furthermore, the production branch contains tags of all previous releases. No development takes place on the production branch, it only serves as a target for merges from release or hotfix branches.

Hotfix branches

If a severe bug is found in one of the releases and fixing cannot be postponed until the next release, a hotfix branch will be forked from the production branch. The bug is then fixed on the hotfix branch and the hotfix branch is merged back both into the production branch for a new minor revision and also into the master branch before it is deleted.

Feature branches

With respect to development, feature branches are the most important type of branches. All development takes place on feature branches. Only when a feature is considered complete and ready for release the corresponding feature branch is merged back into the master branch and deleted afterwards.

9.1.1 Why use Branches

The introduction of the previously described branching guidelines provides many advantages to the KIELER development process. Most important of all is the introduction of feature branches. As all development is done in different feature branches, the master branch is always in a release ready state. This means that a release branch can always be forked from the master branch without the need to remove incomplete features after branching, leading to less overhead in the release process.

The introduction of a production branch allows for easy hotfixing. As the production branch's head always reflects the code of the most recent release, hotfix branches can be forked easily. The reason for forking hotfix branches is to allow for collaboratively fixing issues in a contained environment that can be quality tested without altering the production branch's head. Also, as release maintenance in the KIELER project is only done for the most

recent release, if at all, it is not needed to keep old release branches, because all information is available on the production branch.

9.1.2 Developing Features

When choosing a feature-driven approach to software development and source code management, all developers have to think carefully about how to break down their development plans into small features. If for example the chosen feature chunks are too large and a feature branch stays open for too long, development between two different feature branches can drift far apart and conflicts can occur when the second feature is merged back into the master branch. Therefore, some rules of thumb are defined for developers:

1. Break down development into small features.
2. Regularly merge changes from the master branch into the feature branch.
3. Keep track of feature development by creating tickets in Jira.

These three rules should ensure that development on branches stays transparent for all developers and also that merging back into master should not become too complicated.

Another important point is to think about when feature branches are ready to be merged back into the master branch. As it has been said, the master branch should always be in a release-ready state. Therefore, all feature branches that are to be merged back into the master branch should also contain only release-ready code. To make things clear the following rules must be fulfilled for a feature in order to be considered done:

1. All planned functionality must be implemented and working.
2. The code has to be well commented; especially Javadoc comments have to be complete.
3. The code must not contain any warnings, including FindBugs and Checkstyle warnings. If single warnings cannot be avoided, a comment is to be left in the code.
4. The code should be well covered by unit tests.
5. The code should have been reviewed.

While it is absolutely mandatory for features to fulfil the first three rules, rules four and five cannot be handled as strictly. If a feature cannot be properly tested with unit tests, developing tests may be omitted. Code reviews can also be postponed, if there is currently no spare time for reviews or if the feature is rather small.

9.2 The KIELER Release Process

With the discussed conventions for source code management a lean KIELER release process can be defined. KIELER will be put on a fixed release schedule with releases occurring every three months. In order to match the university's vacation schedule, releases will be scheduled for the last day of March, June, September and for the last work day in December. The release process is then defined as follows.

Three weeks before release

Three weeks prior to the release the project secretary calls the developers to go through their open issues and see if these can be resolved for the next release or have to be postponed.

Also, developers should test their code carefully to prevent bugs from being merged into the release branch. Furthermore, developers should go through the wiki documentation to check if everything is up-to-date. A release page for the new release is created in Confluence to collect release notes of new features for the release.

Two weeks before release

Two weeks before the release, the release branch is forked. Additionally, on the Bamboo server, build plans are created for the new release branch, one continuous plugin build and one nightly RCA build. Both builds inherit all settings from the corresponding build plans of the master branch. Developers are invited to do final testing and bug fixing and also to increment the version numbers of their plugins according to development progress.

The project secretary has to do all technical preparations for the release. This includes preparing download directories for the release, preparing a release repository, etc. The full list of preparation steps can be found in the KIELER documentation.

One week before release

One week before the release final release preparations have to be made. The project secretary keeps a close look on open tickets and reminds developers to fix their remaining open tickets and to do final release testing. If tickets cannot be closed until the release date, the project secretary decides if their contents are added to the *Known Issues* section on the release page and the tickets are rescheduled for hotfix.

One day before release

The project secretary decides together with the developers if any tickets that are still open are rescheduled for hotfix or for the next release. The content of all tickets that are rescheduled for hotfix is then added to the known issues section.

Release day

On the release day, the project secretary first runs the final RCA build and updates the contents of the download and of the KIELER P2 release repository. He also updates the KIELER project website with the new release information, download site, etc. Then, he writes an email to the KIELER announce mailing list to inform users that a release has occurred. This email should also contain the release notes with new features and known issues.

Afterwards, all changes from the release branch are merged into the production branch and a tag is set for the new release. Finally, all changes from the release branch are merged into the master branch so that all bugfixes that were done in the release branch are also put into master. The release branch is deleted afterwards. The KIELER team commences to celebrate[2].

[2] http://trololololololololololo.com/

Chapter 10

Other aspects

In the last chapters we have discussed various aspects of the KIELER development process. All these together form the new KIELER software process. However, there are aspects that have only been mentioned briefly, and other aspects that have not been mentioned at all.

In this chapter we discuss these smaller topics that act as a glue to form a complete software process.

10.1 Communication

In successful software projects, good communication between team members is very important. Especially when the project is very large and developers work individually on their small parts of the software measures have to be taken to ensure that all team members are informed about what is going on in the project.

For the KIELER project, multiple ways of communication have been established. Developers can communicate through the issue tracker when they discuss problems. Also all developers should record their development tasks in the issue tracker to keep everybody informed about who is currently working on what.

Besides the issue tracker there is also the KIELER developer mailing list. While there is not much technical talk on the list, it is actively used by the lead developers to keep people informed about the topics of the weekly meeting and also of their outcome. Furthermore there is a KIELER user mailing list that is mainly used for release announcements.

In order to provide another fast way of communication an instant messaging (IM) server has been installed. Developers are encouraged to log in on this server while they are doing development on KIELER. While not all developers have joined the server yet, the developers that are already using the server are fond of the fast and asynchronous way of communication.

10.2 The weekly KIELER Meeting

The weekly KIELER meetings are another part of the KIELER team collaboration. Having been rather unstructured in the past, measures have been taken to make these meetings more effective.

First of all, the agenda is gathered during the week and announced one day prior to the meeting on the KIELER developer mailing list. This allows for more structured meetings and for participants to prepare for the topics so that a better discussion can take place. Also, lengthy discussions about specialised topics are cut off in order to streamline the meeting.

Furthermore, some regular meeting topics have been introduced. Every week, a short overview on the current project state is given; this includes discussing open tickets and a

summary of administrative topics that arose during the last week. Also every developer summarises his last week's work and explains his development plans for the coming week.

10.3 The Project Secretary

The project secretary has already been mentioned briefly in the former chapters (see sec. 6.5, 7.4, 8.1.2, 9.2). He is responsible for the administrative tasks that arise during development. Similar to a ScrumMaster he makes sure that all project rules are followed. In the past it could be observed that if there is nobody who feels responsible for the project guidelines, compliance of the developers decreases quickly.

The secretary assists the developers in the technical development tasks. For example, he explains the build system and the usage of the source code management system. When there are technical issues that hinder development, he finds solutions together with the development team.

In case of the KIELER project, the secretary is also responsible for the build process (see sec. 5.3). He creates build plans as needed and makes sure that the build server is running fine. Also, he constantly strives for its improvement.

In the issue tracker, the secretary ensures that all developers make proper use of tickets (see sec. 6.5). Therefore, he has to monitor open issues and due dates and check for proper classification; in other words, he has to keep up the order in the issue tracker.

Having a project secretary has shown to be very important for the KIELER project. It frees the developers of administrative tasks, so they can fully concentrate on software development. Furthermore, as the secretary in not part of the development team itself, he can observe the project from a neutral point of view so he can decide without being influenced by development decisions.

Chapter 11

Evaluation

To verify the effectiveness of the described process and also to see if developers are happy with the new guidelines and software, in this chapter we will both learn about the developers's point of view as well as approach the evaluation from a technical point of view.

11.1 Technical Perspective

For the technical evaluation of the KIELER software process we first take a look at the server performance. The information from the system accounting logs show that memory consumption on the project management server is fine. On average, the memory usage lies at about seven Gigabytes, which in peak times when multiple builds are running increases to about eleven Gigabytes. Swapfile usage lies at a stable average of around 350 MB, page swapping does not not occur frequently and is especially not associated with phases of high system load. This means that only inactive memory pages are swapped in favour of filesystem write buffers and read cache. Overall it can be said that, memory-wise, server performance is fine.

Looking at the system load statistics shows that the daily average system load is below one. In peak hours when multiple concurrent builds are running the system load rises to about six. The system load is derived from the average kernel run queue length in one minute: A load of one means that, on average, one process at a time requested CPU cycles during the last minute. As the project management server has eight CPU cores up to eight processes can be run concurrently. Therefore, system performance can be considered as fine.

Regarding free disk space things will have to be improved in the future. Currently filesystem usage of the `/var/atlassian` filesystem is at about 60%, 101 of 180 GB are used. While there is no current danger of running out of disk space, this has to be closely monitored in the future. On the filesystem that hosts the KIELER git repositories the situation is even graver; 96 of 111 gigabytes are in use, this means a relative usage of 96%. Therefore, it is advised for the future to invest into new hard disks to expand disk space as freeing disk space is impossible without losing functionality.

We will now take a look at how the newly installed software tools impact software quality. The CI server has conducted 467 continuous plugin builds since installation and 72% of the builds were successful. Taken into consideration that 67 build failures could be tracked down to flaws in the Maven build process this means that only about 14% of the builds failed due to development errors. On average it took one further build to fix the build errors. That shows that developers are aware of failing builds and take quick responsibility to fix broken builds. The situation with RCA builds, however, can be improved. Only 50% of the nightly RCA builds were successful. Apart from flaws in the build process it can be observed that many build failures result from inconsistencies between Maven and Eclipse metadata. When for example a new plugin is developed, a `pom.xml` file for the new plugin has to be written. If this

step is omitted, this will result in a failure of the headless build as not all dependencies can be resolved. Learning from this, it is advised to better train the developers on build process and also to consider the usage of the *Eclipse M2E* plugin, which can create `pom.xml` files directly from Eclipse.

Now that the build infrastructure has been covered, we evaluate how the installation of Jira has affected issue tracking. Regarding ticket creation, nearly no difference can be observed. While the number of created tickets has risen from 10 tickets in January to 35 tickets in July, it cannot be safely attributed to the migration from Trac to Jira. Rather, it is more likely that the number of generated tickets is highly influenced by how much development in KIELER takes place in general. This is also indicated by the large number of generated tickets in July which was a consequence of the application of FindBugs and Checkstyle on the KIELER sources.

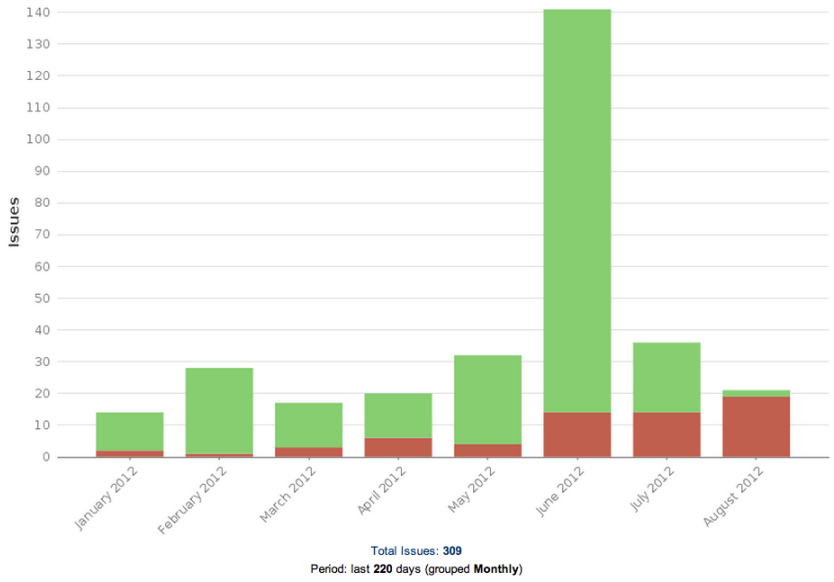


Figure 11.1. Created tickets in Jira

A better indication of the improvements in the issue tracking process is given by looking at the average ticket age. There it can be observed that the average number of days issues were unresolved over a given time period has dropped from around 330 days in the second half of 2011 to currently 160 days. This shows that the developers are more aware of open issues and take more care to fix or at least reschedule them to the indefinite future milestone, which has been left out of the calculation.

11.2 Developers Perspective

To get a feeling about what developers think of the changes in the KIELER software process, the KIELER lead developers were asked another series of questions. All interviewed developers were asked the following set of six questions:

1. What is your opinion on the new build process?

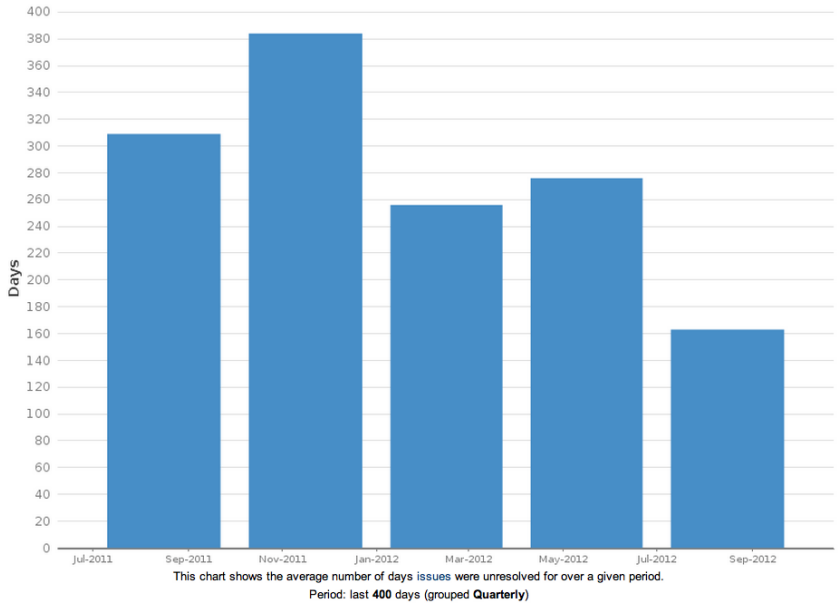


Figure 11.2. Average KIELER ticket age

2. What is your opinion on issue tracking in KIELER?
3. What is your opinion on the code review process?
4. Do you think the KIELER documentation has improved or deteriorated?
5. Do you think that the new process helps to get a better overview over the project as whole?
6. Did your work load change after the implementation of the new process?

Overall, the impression that the new process has left on the developers was rated positively. The detailed answers are described below.

The Build Process

Regarding the build process, all developers agreed that it works well. Most developers praised the clear and intuitive web interface of Bamboo. They all agreed that information on the build status can be gathered easily and also notification on failed builds happens promptly. One developer emphasised that notifications via instant messaging is an unobtrusive way of information delivery that does not need regular checking of email. One developer suggested that failed builds should be discussed at the daily meeting to keep the developers even more informed. Other things that were mentioned positively were the automatic creation of builds for branches, the automatic assignment of responsibility for failed builds, and the integration of Bamboo with the other project management tools. There was no mention of negative aspects. However, one developer did not realise any advantages compared to the old build process and one developer suggested that it should be made clearer why builds fail.

Issue Tracking

Asked about the issue tracking process, all developers agreed that the new issue tracking software Jira provides many benefits compared to Trac. All developers agree that Jira's usability is very good. Chart generation and the activity stream help to keep track of the development progress. Several developers mentioned the integration with the other project management tools as positive. Especially the integration with code reviews in Crucible and source code in Fisheye is considered good. Other positive things that were mentioned included the possibility to get information on status changes of certain tickets by adding them to a personal watchlist and the easy possibility of adding comments to tickets. No suggestions for improvements have been made.

Review Process

The code review process was also evaluated positively. All developers agree that the process is working fine. One developer would like to have more IDE functionality in Crucible, such as the possibility to jump to declarations of methods. However, all developers like the asynchronous review process Crucible allows so they can continue reviewing whenever there is some spare time left. Also, the developers like that classes are now reviewed as whole, contrary to the old process where only parts of classes were presented by the author. No further suggestions on the review process have been made.

KIELER Documentation

All developers agree that the KIELER documentation has improved. Through the restructuring that coincided with the migration from Trac to Confluence, documentation has become clearer and it has become much easier to get information on particular topics. All developers like editing wiki pages in Confluence, with most developers even thinking that they are going to write more documentation due to that fact. However, most developers think that the KIELER documentation should be improved further.

Project Overview

As the interviews were only conducted with the lead developers, they all had a good understanding of the KIELER infrastructure before. However, all developers stated that gathering information about the current development status has become easier. Most developers mentioned the configurable Jira dashboards that allow to group information widgets on a configurable website.

Work Load

Regarding work load, all developers stated that, as everyone is doing code reviews now, work load has risen slightly. However, no developer sees this as problematic. All developers expressed that they expect work load to decrease slowly due to the new process. One developer said that the appointment of the project secretary freed him from administrative duties so that he can now concentrate more on research and development.

Chapter 12

Conclusion and Future Work

In this thesis, a software process for the large academic software project KIELER has been designed. We have learnt that development in academic project differs from commercial software development in many ways. Therefore, the application of well-known and well-researched development processes is not feasible. The process that has been defined is designed to be lightweight as possible and to allow as much freedom as possible for the researchers without losing control over the complexity.

The foundation of this process is a strong set of tools that both help the developers with their daily work, and also easily allow to get an overview about the current project state. Supported with a small set of rules and a project secretary who makes sure that all rules are obeyed and also takes care of software administration, development in the KIELER project is well on its way. The role of the project secretary has been shown to be most important over the course of writing this thesis. It could be observed that if the project secretary did not take enough care, code review sessions for example were not scheduled. This might be due to the fact that the process is rather new and developers are not enough used to it yet, but nevertheless the importance of a project secretary cannot be stressed enough.

Looking into the future there is much work yet to be done. First of all, the process proposed in this thesis should evolve, adapted to the project's changing needs. Code reviews for example are currently scheduled once per week, and also they have to be enforced and scheduled by the secretary. This process is meant to evolve over time to a review culture in which reviewing source code becomes a self-evident part of software development, only supported but not closely monitored by the project secretary. In order to establish this good development culture, focus has to be put on the training of new developers. While there is currently a written guide for new developers kept in the project wiki, it is advisable for the future to implement some kind of coaching program where new developers are guided through KIELER's coding and project management culture.

Another part of the process that needs further work in the future is conducting software releases. While this thesis proposes a release process, it has not yet been proven to work well. After future releases, the process will have to be evaluated and improvements will have to be discussed. Also the frequency of software releases has to be discussed; this thesis suggests a three-monthly schedule, but a more flexible schedule that reflects the fluctuation of development done in KIELER could be considered.

Considering software tests, there is also work left to be done. While this thesis emphasises the importance of automatic testing to improve build stability and to prevent regressions, no actual guidelines on testing have been introduced. To help developers with testing, the implementation of testing frameworks to test for example graph layout or the various simulators in KIELER is advisable. Also a set of guidelines and test examples would help new developers to get ideas on how to implement software tests in KIELER.

From the technical point of view, there is also some space for improvement left. Looking at the build process, software testing could be improved. While integrating JUnit tests of non-UI

enabled plugins works fine, testing plugins that depend on the Eclipse UI still poses some problems that have to be investigated further. Also there are many Maven plugins assisting in various stages of the development process that have not been fully explored. In this thesis only plugins that were essential for a successful build have been enabled.

Further work could also be done on the structure of KIELER itself. As we have learnt, KIELER is a huge project in the world of Eclipse Rich Client Applications. Currently there is an ongoing discussion amongst the lead developers about the possibilities of breaking down KIELER into multiple smaller projects. While this is not directly connected to the software process, a split of KIELER would mean that development is done in smaller teams. The described process can be applied to smaller teams; however, when working with smaller teams new problems, but also new possibilities will arise. For example even when the project is split the newly founded projects will depend on each other. This means that communication between the smaller projects has to be organised to ensure that all parts interoperate without problems. A benefit of smaller independently developed subprojects would be an easier release process. As the projects then would be smaller, conducting releases would also be an easier task because in smaller projects it it less likely that the whole release is blocked by severe bugs, that are discovered shortly before the release date.

This thesis mainly dealt with the development process. Software engineering also puts an emphasis on the software design process. The explicit introduction of both design and architecture patterns could improve the software design of KIELER. While design reviews are mentioned as part of this thesis, guidelines are far from being fully developed yet. The implementation of template pages in Confluence for the documentation of design review would for example help to establish more structured reviews. There are also large parts of KIELER that have not undergone a design review yet. Constantly design reviewing unreviewed parts of KIELER would lead to a more established review culture and could help to establish design review guidelines for the KIELER project.

Finally, it has to be stated once more that the KIELER software process relies heavily on the project secretary. He has to observe the whole process, striving for its constant improvement in coordination with the developers. He should also look for new methods of improving software development by investigating new software tools that support the development process.

Bibliography

[Atl12] Atlassian Software. Bamboo Administration Guide. online, 2012. https://confluence.atlassian.com/display/BAMBOO/Administering+Bamboo.

[BA04] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[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*, pages 2002–04, 2001.

[Bec99] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[BJ87] F.P. Brooks Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[BJS+08] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318. ACM, 2008.

[Boe88] B.W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[BVGW10] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 291–300. Citeseer, 2010.

[CMPS06] J. Casey, V. Massol, B. Porter, and C. Sanchez. Better builds with maven, 2006.

[Coc05] A. Cockburn. *Crystal clear: a human-powered methodology for small teams*. Addison-Wesley Professional, 2005.

[Coh04] M. Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. 1972 ACM Turing Award Lecture.

[DMG07] P. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[DS90] P. DeGrace and LH Stahl. Wicked problems. *Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms, Yourdon*, 1990.

[Fag76] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[Fow06] Martin Fowler. Continuous integration. online, 2006. http://www.martinfowler.com/articles/continuousIntegration.html.

[FvH08] Hauke Fuhrmann and Reinhard von Hanxleden. On the Pragmatics of Model-Based Design. In *Monterey Workshop*, pages 116–140, 2008.

[HK07] D. Huizinga and A. Kolawa. *Automated defect prevention: best practices in software management*. Wiley-IEEE Computer Society Pr, 2007.

[IBM06] IBM. Eclipse Platform Technical Overview. online, 2006. http://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[JBH+10] Johannes Jacop, Marcel Bruch, Stephan Herrmann, Jens von Pilgrim, Mirko Seifert, Claas Wilke, Birgit Demuth, Matthias Hanns, Simon Kronsreder, Stanislav Elinson, and Maximilian Kögel. Eclipse & academia. *Eclipse Magazin*, 6.10, October 2010.

[JD03] J.N. Johnson and P.F. Dubois. Issue tracking. *Computing in Science & Engineering*, 5(6):71–77, 2003.

[Loe09] J. Loeliger. *Version control with Git*. O'Reilly Media, 2009.

[Lou06] P. Louridas. Static code analysis. *Software, IEEE*, 23(4):58–61, 2006.

[LS80] BP Lientz and EB Swanson. Software maintenance management, 1980.

[NR69] P. Naur and B. Randell. Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th to 11th october, 1968. Nato, 1969.

[OSG07] OSGi Alliance. *About the OSGi Service Platform*, Juni 2007. http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf.

[RG06] P.C. Rigby and D.M. German. A preliminary examination of code review processes in open source projects. Technical report, Technical Report DCS-305-IR, University of Victoria, 2006.

[RNHL99] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software practice in the Ptolemy project. Technical report, Gigascale Semiconductor Research Center, April 1999. http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/ptII/doc/coding/sftwareprac/software-practice.pdf.

[Roy70] Winston W. Royce. Managing the Development of large Software Systems. In *Proceedings of IEEE WESCON*, volume 26, pages 1–9, 1970.

[S+95] K. Schwaber et al. Scrum development process. In *OOPSLA Business Object Design and Implementation Workshop*, volume 27, pages 10–19. Austin, TX, 1995.

[SB01] K. Schwaber and M. Beedle. *Agile software development with Scrum*, volume 18. Prentice Hall, 2001.

Appendix A

Questionnaires

- General questions
  - For how long have you been part of the development team?
  - On which part of KIELER are you working?
  - How did you learn about KIELER?
  - Should KIELER be used more or less for lectures?
  - Do you have any ideas on how to get people to join the development team?
  - Do you have any experience in other software projects?
  - How did you feel after joining the KIELER team?
- Project organisation
  - How would you rate team communication?
  - Do you have ideas how communication could be improved?
  - What do you think about the weekly developer meetings?
  - Are you happy with how the meetings are run?
  - Is there anything that could be improved regarding meetings?
  - Do you think that KIELER is properly documented?
  - Are there any parts of KIELER whose documentation should be improved?
  - How do you document your work?
  - What do you think about Javadoc documentation?
  - How do you use the Trac system?
  - Is the presentation of information in Trac fine?
  - Do you like working with the KIELER Trac wiki?
  - If there was a more convenient wiki system, would you use it more?
  - Are the contents of the KIELER wiki up-to-date?
  - How could the contents of the KIELER wiki be improved and kept up-to-date?
  - Do you have an overview about currently open Trac tickets?
  - How do you use the issue tracker?
  - How would you rate the usability of the issue tracker?
  - Do you think that the KIELER developers make enough use of the issue tracker?
  - Do you have any ideas on how people could be encouraged to make more use of the issue tracker?

- Do you think the issue workflow is sufficient?
- Do you think that issues are timely resolved?
- Do you have any ideas on how to speed up processing of tickets?
- Would you like to have a connection between the IDE and the issue tracker?
- What happens during a release?
- Which problems arise during the release process?
- Do you have any suggestions for improvement of the release process?
- Is there a fixed release cycle that you know of?
- If there is an upcoming release, do you know how much work is left until the release can be conducted?
- Do you think that other developers know how much work is left?
- If a release has been made, is there any hotfixing done afterwards?
- Would you consider this useful?
- Could you describe the current KIELER build process?
- What do you think about the build process?
- How do you use the CI server?
- Do you think the CI tool has a clear interface?

- Sources/Development
  - How did you experience the switch from SVN to Git?
  - Which features of Git do you use?
  - Do you make use of branches?
  - Do you think the KIELER source tree is well-structured?
  - Would you suggest any improvements to the structure of the source tree?
  - How do you test your new features?
  - How do you ensure that your development does not break other parts of KIELER?
  - In which form should software tests be introduced for KIELER?
  - Do you have an idea how software tests could be integrated into the build process?
  - How would you rate the code quality of KIELER?
  - Are the KIELER coding conventions observed?
  - Do you ensure that all Checkstyle warnings are removed before you check-in new source code?
  - Does the same hold for FindBugs warnings?
  - Do you think the KIELER sources need overhauling?
  - Which parts of KIELER should be reworked?
  - Do you regularly rework your own code?
  - How could the overall source code quality be improved?
  - Do you know anything about pattern oriented design?

- ▻ How could the KIELER software design be improved?
- ▻ What is your opinion on code and design reviews?
- ▻ Are there enough design reviews being conducted?
- ▻ Are there enough code reviews being conducted?
- ▻ Do you know any reasons why only few code and design reviews are conducted?
- ▻ Do you think the quality of KIELER has improved or deteriorated?
- ▻ What do you think about the current KIELER review scheme?
- ▻ Do you have any suggestions to improve the review process?
- ▻ Do you know any well-described software processes?

▻ Closing questions

- ▻ Do you have any other remarks on things working good in the KIELER project?
- ▻ Do you have any other remarks on things working bad in the KIELER project?
- ▻ Do you have any other ideas on improving the KIELER development process?

Appendix B

Documentation from the KIELER wiki