The Influence of Software Maintainability on Issue Handling

Master's thesis

Bart J.H. Luijten

The Influence of Software Maintainability on Issue Handling

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bart J.H. Luijten
born in Margraten, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Software Improvement Group
A.J. Ernststraat 595-H
1082LD Amsterdam
The Netherlands
www.sig.nl

©2009 Bart J.H. Luijten. All rights reserved.

The Influence of Software Maintainability on Issue Handling

Author: Bart J.H. Luijten
Student id: 1174622
Email: b.j.h.luijten@student.tudelft.nl

Abstract

Ensuring maintainability is an important aspect of the software development cycle. Maintainable software will be easier to understand and change correctly. The Software Improvement Group (SIG) has developed a method to measure a software system's maintainability based on well-known code metrics. In this thesis we explore the relationships between this maintainability measure and the properties of issues reported for a project. We also describe the repository extraction tool that we built for this purpose.

We investigate a number of basic issue properties and show that we cannot draw conclusions based on these metrics. Two visualisation techniques are used to gain a more detailed understanding of the issue handling process. The Issue Churn View shows quantitative changes in the open issues for a project and can be used to show the changes in activity in an issue tracker. The Issue Lifecycle View provides a more detailed view of the issue handling process and shows the age composition of the issues, as well as simultaneous events on multiple issues.

The SIG quality profile model is applied to issue trackers, resulting in a model that allows us to compare systems based on the speed with which issues are resolved. By comparing the ratings from this model to system maintainability, we conclude that there is significant correlation between system maintainability and defect resolution time. The most correlated system properties are unit size and complexity and module coupling.

Thesis committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Dr. ir. J. Visser, Software Improvement Group B.V.
Committee member:	Dr. ir. A.J.H. Hidders, Faculty EEMCS, TU Delft

Preface

This thesis represents the work I have done as a Master's thesis project at the Delft University of Technology. The work was performed between September 2008 and December 2009, under the supervision of Dr. Andy Zaidman and Dr. ir. Joost Visser, at the Software Improvement Group.

I am very grateful to my direct supervisors, Andy and Joost, for sticking with me and supporting me. They got me back on track when work was slipping and provided valuable feedback on my work and thesis. Special thanks go to Oliver Burn, lead developer of Checkstyle, and Linus Tolke, lead developer of ArgoUML, for providing valuable feedback on my graphs.

Many thanks also go to Miguel, Christiaan, Zé Pedro, Tiago and the other SIG researchers and (ex-)interns, for helping me with my work and giving useful comments and suggestions. Furthermore I would like to thank everyone else at the SIG for giving me the chance to do this project and for the friendly and helpful, yet challenging environment.

Finally, thanks go to my parents for the support they gave me throughout my education at the TU Delft.

Enjoy reading my thesis,

Bart J.H. Luijten
Amsterdam, the Netherlands
January 20, 2010

Contents

List of Figures

List of Tables

List of Abbreviations

SIG Software Improvement Group

SAT Software Analysis Toolkit

ISO International Organisation for Standardization

VCS Version Control System

ITS Issue Tracking System

LOC Lines of Code

ICV Issue Churn View

ILV Issue Lifecycle View

ITT Issue Throughput Time

Chapter 1

Introduction

Assuring maintainability is an important aspect of the software development process. A large part of the cost of the software product lifecycle comes from maintenance [11, 22], after the initial product has left development. Therefore, it is of great importance to construct software that is as easily maintainable as possible.

To aid the construction of maintainable software, a large number of software metrics have been developed. Well-known software metrics such as McCabe's cyclomatic complexity [28] or the object-oriented metrics suite by Chidamber and Kemerer [11] are often used to measure a system's complexity and, through that, its maintainability. The Software Improvement Group (SIG) has created a model which assigns a maintainability rating to a system, using a number of well-known software metrics [21]. This rating is also based on the maintainability characteristics defined in the ISO 9126 [23] software quality standard.

A large part of the maintenance effort for a system is spent resolving issues that arise during development and use of the system. Because of this, we expect there to be a relationship between a system's maintainability and the efficiency and effectiveness with which issues are resolved. Intuitively, an issue for a system that is extremely complex and badly maintainable will require much more effort to be resolved than for a system that is more maintainable. This thesis will verify the existence of such a relationship.

The process of reporting and resolving issues for a system is often handled through the use of an Issue Tracking System (ITS). In an Issue Tracking System (ITS), all issues affecting a system are tracked during the issue handling process. Because of the dependency between maintainability and issue handling, we suspect the maintainability of a software systems to be reflected in the issue tracker associated with that system. In particular, we suspect that issues will take more effort and time to be resolved for systems that have lower maintainability. This study attempts to discover the relationships between the SIG maintainability ratings and the issue properties of systems.

Section 1.1 will introduce and discuss the research questions we seek to answer. Then section1.2 outlines our approach to answering these questions. A brief description of the company this research was conducted for is given in section 1.3. Finally, section 1.4 explains the contents of the rest of the thesis.

1.1 Problem Statement

As a research project, we will investigate software systems from a maintainability point of view. We will investigate the evolution of software metrics over time, with a focus on code maintainability. This will be compared to the changes in issue metrics over time, with a focus on issue resolution effort. A correlation between the two would be interesting, since that would indicate the particular metric involved might be a predictor for issue handling effort. The research goal is to determine how an issue tracking system reflects the maintainability of the software source code it is related to.

We would like to investigate this relationship both in a qualitative and quantitative manner. We will both compare issue handling between projects, and investigate the changes in issue handling within projects. The following research questions capture our intent:

RQ1 : Can we distinguish differences in issue metrics between different projects or project phases?

RQ2 : How does production code maintainability affect the issues reported for a project?

The answers to these research questions will give us a better understanding of the effects software maintainability has on issue handling. This in turn will allow us to determine which maintainability aspects are likely to result in reduced issue handling effort when they are improved.

By using the SIG maintainability rating as a measure of software maintainability, we also provide validation for the use of this rating. If a relation is found with the issue handling, this will strengthen the confidence in the rating as an indicator of maintenance effort. This can help the SIG in giving their clients advice on how to improve their software, again to minimize maintenance effort.

1.2 Approach

To answer the research questions posed in the previous section, we will examine the Version Control System (VCS), ITS and source code metrics of a number of open source software projects. To obtain and store this data, we will first examine a number of existing efforts for opportunities of reuse. If necessary, we will implement a repository extraction tool. Once we have selected our subject system, we will use the extraction tool to obtain the necessary ITS and VCS data.

Ideally, we would be able to directly relate an issue to the source code fragments used to resolve it. This would allow us to get a very detailed picture of code metrics versus issue metrics. Unfortunately, the experiment we did in the context of our initial literature study has revealed that it is difficult to automatically construct an accurate relation of issues to source code entities. Issue numbers are not systematically and recognisably recorded in change messages. Since this introduces significant bias into the dataset [7], we decided not to use this approach. Instead, we will pursue a more high-level approach, using system-level metrics.

The SIG Software Analysis Toolkit (SAT) will be used to get source code metrics for the subjects. To quantify the issue data, a number of metrics will be defined on the ITS, along with hypotheses on how these will be influenced by the differences in maintainability between subjects.

In order to validate the hypotheses, we will use common statistical tools to determine the correlations between maintainability ratings and our issue metrics. The result of this will show us how the maintainability and issue metrics are related and allow us to answer our second research question.

For the first question, we plan to investigate the issue trackers of our subject systems in more detail. We will develop a visualisation that captures the changes of the issue metrics over time. By comparing this between systems, and observing trends within systems, we should be able to discover how issue handling is different between systems and development phases. To validate the visualisation, we will discuss our observations of a system with the developers.

1.3 Context

The Software Improvement Group (SIG) is a company specialized in software quality consultancy. The SIG applies static source code analysis techniques to assess the technical quality of software products based on hard facts. Source code measurements are used to provide risk assessment, monitoring and quality certification services. These services are available for a wide range of system types, from ancient mainframe applications to modern Java systems.

The SIG has developed a method to assess the maintainability of a software system, relative to other systems. Using this method, a so-called maintainability rating can be assigned to a system. To complement this, the SIG provides certification services, signifying that a system achieves a certain level of maintainability.

By gaining insight into the relationship different aspects of software maintainability have to the process of handling issues, the SIG can give more confident recommendation to its customers regarding reduction of issue handling effort. Understanding how maintainability impacts issue handling helps in explaining which aspects of a software system should be changed in order to minimize future issue handling effort.

1.4 Organisation

This thesis is structured as follows. First, chapter 2 will discuss related work and background information. Next, chapter 3 shows how we obtain and store our subject data. A detailed description of the initial experiments can be found in chapter 4. Following that, we will discuss the visualisations that we developed in chapter 5. Chapter 6 shows how we applied SIG technology to quantify the issue handling process, and how this corresponds to source code quality. Finally, chapter 7 draws conclusions and contains suggestions for future work.

Chapter 2

Related Work

This chapter provides the rest of our work with a frame of reference in literature. First, section 2.1 introduces the concept of metric validation. Sections 2.2 and 2.3 discuss version control and issue tracker mining respectively. Finally, section 2.4 shows what kinds of repository data are currently available as publicly accessible databases.

2.1 Metric Validation

Metric validation is the field of study concerned with determining the values of software metrics, i.e. determining whether these metrics measure useful aspects of the system under study. Many metrics capture aspects of a system that are in some way related to the difficulty of understanding it, or its complexity. Examples are McCabe's cyclomatic complexity [28], coupling metrics or even size metrics like Lines of Code (LOC). A common intuition is that these metrics act as an indicator for where faults are most likely to occur. Studies on metric validation, such as those discussed below, attempt to find statistical or causal relations between metric values and the occurrence of faults.

A well-known set of metrics is that by Chidamber and Kemerer [11]. Gyimothy et al. [20] study these metrics within the Mozilla project[1]. They assign bugs in the ITS to source code in a specific release by examining patch files submitted with a bug. Using logistic regression analysis, they show that the Coupling Between Objects metric is the best predictor for fault occurrence, with precision and recall of just under 70%. However, due to the necessity of having patches available, their approach only takes a small percentage of bugs into account, which compromises generalizability. A further experiment using machine learning models (decision trees and neural networks) did not improve these results.

Another evaluation of the Chidamber and Kemerer metrics suite was later done by Briand et al. [10], who also examine a large number of metrics defined by others. They perform logistic regression techniques to determine how the metric values are related to the presence of faults. As a dataset, they use systems developed by students in a computer science course, with the faults determined by an independent team of professionals. Their best fitted model consists of four coupling and three inheritance measures, with a precision

[1] `http://www.mozilla.org`

of 84% and recall of 94%. This result is consistent with others in placing emphasis on coupling metrics above size and complexity, but the use of student projects, instead of full-sized commercial or open source systems, constitutes a threat to the validity of this study. Interestingly, the often-used McCabe complexity [28] is not included in this analysis. A number of later works have replicated the validation of the Chidamber and Kemerer metrics, using various techniques such as threshold models [4] and neural networks [34].

Fenton and Neil [14, 15] take a different approach, asserting that traditional methods of metric validation are using metrics in too much isolation, without regard for unknown causal effects between them. They also suggest that many studies make use of inappropriate statistical methods and that there is too much emphasis on the presumed relation between pre- and post-release defects. As a solution to this, they propose the use of Bayesian Belief Networks, which can take into account both empirical data and domain expert knowledge, along with uncertainty in causal relations. The example provided by the authors has unfortunately not been validated, so it is unknown how accurate this proposed approach actually is.

In our work, we attempt to find validation for the SIG maintainability ratings, which are derived from static source code metrics. Instead of using a dataset consisting of just one system, or built from student projects, we use multiple large open source projects as research subjects. We distinguish ourselves by using a high-level, aggregated approach to issue mining, so that we are not dependent on the presence of patches or issue-to-source links.

2.2 Version Control Mining

During the development of a software project, software repositories are often used to keep track of various software artefacts, such as source code, tasks or related e-mails. A repository is defined by Bernstein and Dayal [5] as "a shared database of information about engineered artefacts produced or used by an enterprise." In a software engineering context, usually every type of artefact is stored in a separate repository, such as a Version Control System (VCS), Issue Tracking System (ITS), or mailing list archive.

A software repository provides a central database where information regarding the development process is stored. As such, it provides many opportunities for research on software evolution through the use of repository mining techniques [24]. Even though VCSs "are not designed to answer questions about process properties" [29], a number of research directions have emerged that make use of these repositories. An overview of many different approaches to software repository mining is given by Kagdi et al. [24]. This section discusses some examples of the major directions in repository mining with respect to software evolution.

The use of VCSs for co-change analysis was first done by Ball et al. [3], who visualised the changes in a commercial software project. Using clustering analysis, they were able to show which files often changed together. Frequent co-changes can indicate a hidden, logical dependency between parts of the system. This thought was also explored by Gall et al. [17], who examined project release histories for co-changing subsystems.

The notion of a co-change *graph* was coined by Gall et al. [18]. Such a graph contains the modules in a system (such as classes or files) as nodes, and edges that indicate whether two modules have been changed together. Beyer and Noack [6] develop the co-change graph further and show that a visualisation of such a graph can lead to good insights into internal dependencies of a system. Co-change analysis can also provide a view of the testing practices used in a project, as Zaidman et al. [35] and Lubsen et al. [27] show.

Eick et al. [13] take a different view by looking for evidence of quality degradation in a VCS, the so-called *code decay*. They show that the average number of files involved in a single change increases during the evolution of the system, indicating increasing levels of coupling and thus decreasing quality.

2.3 Issue Tracker Mining

In addition to the VCS, many software projects make use of an ITS to keep track of bugs, tasks and other issues that arise during development. A popular issue tracker, often used in open source projects, is Bugzilla[2]. Commercially used systems include Jira[3] and Rational ClearQuest[4].

All ITSs store similar data about issues. An issue as present in Bugzilla and most other ITSs contains some basic properties like an identifier, summary and current status [12, 16]. In addition to these basic properties, an ITS will often keep track of when an issue has been submitted and what changes (e.g. status updates) have been made to it since then. This can be used to compute additional derived properties that might be more useful to research. Hooimeijer and Weimer [22] use many such properties, including the time taken to resolve an issue and the number of issue reported on the same day. Sandusky et al. [31] take this a step further and construct a network of relations between issues, based on commonalities between them. Examples of such relationships are reports with the same developers assigned to them, or reports that are duplicates of one another.

Most of the existing work which makes use of an ITS is focused on bug prediction or prevention. Examples of this are the works of Ratzinger et al. [30] and Graves et al. [19]. Both attempt to predict the amount of bugs affecting a piece of software using the corresponding source and fault history. Graves et al. [19] developed statistical models that predict which properties of the change history predict future faults in software. Interestingly, they found that the major influence is the number of previous changes to a module, and that including complexity in the model does not improve results. Ratzinger et al. [30] uses only evolutionary metrics, but confirms that it is possible to build a good predictor with these.

Zimmermann et al. [36] examine the transferability of bug prediction models between projects. Out of the 622 project version combinations they tested, an astonishingly low 3.4% was able to reach their 75% cross-prediction accuracy threshold. Further investigation allows them to formulate a set of guidelines on how to select a project that will provide

[2] `http://www.bugzilla.org/`

[3] `http://www.atlassian.com/software/jira/`

[4] `http://www-01.ibm.com/software/awdtools/clearquest/index.html`

maximum prediction accuracy for a given target project. Unfortunately, Zimmermann et al. [36] make no claims on the usability of a general model, trained on all subject systems.

It is also possible to attempt to predict bugs in new or changed code by identifying the past changes that introduced a bug [32, 26, 1, 25]. This is done by examining the change log messages for indicators that a bug was fixed and assuming that the change that last touched the same code will have introduced the bug. Future changes that are similar in nature, as measured through complexity, size or other metrics, could also be faulty. Even though this seems to be a rather strong assumption, prediction accuracies of 35% [25] to 60% [1] has been reported. A similar approach is used by Boogerd and Moonen [9], using coding standard violations as a code metric. They found a set of 10 coding rules that are significant fault predictors.

A problem with the previously discussed work is the dependence on available links between VCS and ITS. In order to determine the source code impacted by an issue, many authors use issue identifiers present in commit logs, including [36, 16, 30, 32, 26, 1, 25]. Both Ayari et al. [2] and Bird et al. [7] investigate the validity of this approach. Both show that a large amount of the issues present in the ITS are not traceable to source code. Bird et al. [7] demonstrate that this leads to a bias in test results. This is a major threat to the validity of works adopting this method.

In order to avoid the mentioned threat to validity, our work will not make use of VCS-ITS links. Rather, we look at the total corpus of issues that are reported for a product release or a time period. Furthermore, we are not directly interested in predicting new issues, but in the influence of a system's measured maintainability on issue solving. Lastly, where much related work uses a relatively small number of subject systems, often just one or two, we use a larger number of subjects, to minimize bias towards one system or application area.

2.4 Software Metric Databases

Since the SIG does not yet possess a collection of issue metric evolution data, we examined a number of existing projects that aim to provide such a database. All of these are publicly accessible and focus on open source software.

2.4.1 SQO-OSS

SQO-OSS[5] provides an open-source platform that aims to provide an easy-to-use way of analysing software repositories. Their tool, called Alitheia, can import VCSs, ITSs and mailing list archives. No metrics are calculated by the tool, but it provides a plug-in architecture that allows users to implement their own metrics calculators.

We aimed to use this platform for its repository extraction capabilities, but we had severe problems in using it for even relatively simple examples like Checkstyle. At the time we evaluated Alitheia, the demonstration website was off-line. We did manage to run the software, but did not succeed in analysing the Checkstyle project due to unexpected prob-

[5] `http://www.sqo-oss.org`

lems in the extraction of the Subversion repository. Since the available documentation was very limited, we decided not to use this software.

2.4.2 Ohloh

Ohloh[6] tracks VCS commits and determines language composition of projects. The Ohloh site allows tracking projects for commits and viewing graphs of the LOC size. Other than this, no product or issue metrics are provided.

2.4.3 FLOSSMetrics

The goal of the FLOSSMetrics[7] project is to provide a database with metrics and other information about open source projects. The project approaches this goal by creating a database of open source VCS, ITS and mailing list dumps. However, there are not too many product metrics, beside LOC and McCabe complexity, currently available. FLOSSMetrics provides some tools for extracting data from SourceForge, but we were not able to run these for our own purposes.

2.4.4 FLOSSmole

The FLOSSmole project[8] is similar to FLOSSMetrics, in that it aims to build a database of data on open source projects. The actual data collected is however more of a process-oriented nature. FLOSSmole handles a number of well-known software forges, including SourceForge and Freshmeat. It contains data on developer team size, target operating systems, programming languages and other non code related data. Because FLOSSmole does not store product or issue metrics, we decided not to use this database.

[6] http://www.ohloh.net
[7] http://www.flossmetrics.org
[8] http://ossmole.sourceforge.net

Chapter 3

Repository Model and Tools

In order to obtain the data necessary to answer our research questions, we constructed a Java tool that allows us to capture ITS information from available projects. The tool includes a general data model, which can store the needed data from different issue trackers in a unified fashion. We created a Java program that reads repository dumps into this model.

To give an overview of what data needs to be available in our model, we explain what data a typical issue tracker contains in section 3.1. Section 3.2 lists existing efforts to make ITS data available for research. After that, section 3.3 will show the data model that we used for our work, and section 3.4 describes the corresponding tool and its use.

3.1 Issue Tracking Systems

An Issue Tracking System (ITS) is a software system that is used to manage the submission and solving of issues that arise during the lifecycle of a software project [24]. Issues represent work units that need to be performed on the project, such as adding a feature or fixing a defect. Most ITSs allow the users to distinguish between different types of issues, often between defects (or bugs, faults) and tasks (new features, enhancements, proposed patches). Issue trackers that only or primarily deal with defects, such as Bugzilla[1], are often called bug trackers.

The ITSs represented in our dataset are Bugzilla, Issuezilla (a modified Bugzilla), Jira[2] and the SourceForge tracker[3]. All ITSs we studied share a similar set of properties which they store for each issue. Since we did not know which of these would be ultimately useful for our research, we designed our model to capture as much of these as possible. The properties commonly stored by issue trackers are [16, 12]:

ID : A unique identifier for the issue. This can be used to refer to the issue report from change log messages.

Type : Indicates what kind of issue this is, either *defect*, *enhancement*, *patch* or *task*.

[1] `http://www.bugzilla.org`
[2] `http://www.atlassian.com/software/jira/`
[3] `http://sourceforge.net/`

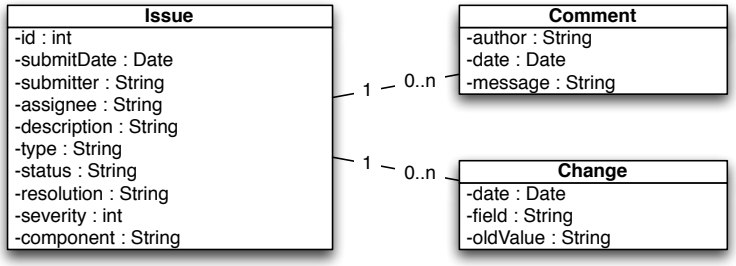

Figure 3.1: Typical issue data model

Status : The current status of the issue, either *new*, *assigned*, *resolved* or *closed*.

Resolution : For closed issues, the reason or solution for closing it, either *none*, *fixed*, *invalid*, *wontfix* or *duplicate*.

Component : The component of the software that is thought to be affected by this issue.

Description : Short and longer descriptions of the problem, often used to include steps to reproduce the problem.

People : The people that are involved with this issue, including the reporter and developer assigned to fix the issue.

Severity : A priority rating for the issue.

In addition to these properties, developers can always place comments on issues, in order to discuss the issue. All studied trackers also keep a history of changes to issue properties. This can later be used to reconstruct the issue lifecycle. See figure 3.1 for an overview of the data commonly stored for each issue.

Note that there are four distinct issue types that are recognized by the typical ITS. A *defect* represents a problem or error in the source code that needs to be resolved, whereas the other three types represent some form of addition or improvement to the source code that is not problematic. Because of this difference, we will use only two categories for the rest of this thesis: defects and tasks. Tasks are the issues marked as *enhancement*, *patch*, or *tasks*, whereas defects are those marked as *defect*.

Typically, an issue does not contain information identifying the source code that contains the resolution for it [24]. In fact, as we have seen in section 2.3, it can be difficult to recover this information reliably.

As for the change history of an issue, the ITSs we used as sources store this in a similar way too. This approach is also pictured in figure 3.1. For each issue, a series of changes is kept, each of which contains three main properties. First, the date of the change, then the name of the field that was changed and lastly the old value of that field. In effect, this means that the stored state of an issue is the current one. The history of an issue can be reconstructed by following (i.e. removing) the changes in reverse order.

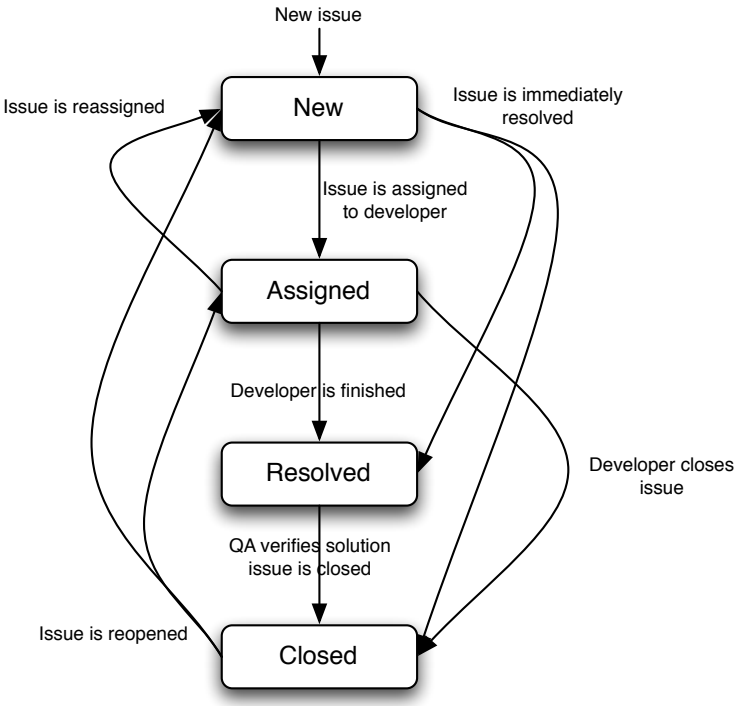

Figure 3.2: Typical issue lifecycle

Each reported issue will follow a basic lifecycle from the moment it is reported. This lifecycle is explained in figure 3.2, which is a simplified version of the Bugzilla lifecycle[4]. Bugzilla includes a *verified* state, which we merged with the *closed* state. The *unconfirmed* and *reopened* states in the original model were merged with *new*. These modifications simplify the model, while increasing compatibility with that of other issue trackers.

The ideal path is straight from top to bottom, through *new*, *assigned* and *resolved* to *closed*. Due to differences in issue handling between projects, sometimes some of the states are skipped. In case an issue does not necessarily need to be assigned to a developer, or is immediately decided to be invalid, the *assigned* state can be skipped. The *resolved* state can be skipped when a project does not have a defined fix verification process, or when the *closed* and *resolved* states have the same meaning within a project.

It is important to note that the part of the lifecycle we are most interested in, the actual solving through modifying source code, is represented by only one step (from *assigned* to *resolved*). There are no further distinguishing phases here in any of the studied ITSs. This means much of the detail of this phase is invisible to the issue tracker, which could obscure the influences of the actual source code that we are looking for.

[4] `http://www.bugzilla.org/docs/tip/en/html/lifecycle.html`

3.2 Existing Efforts

A number of previous efforts aim to provide a set of issue data for research purposes. We examined a number of these efforts to assess their usefulness for our research.

The PROMISE dataset [8] contains data suitable for fault prediction. It consists of donated data for various programs and projects, mostly anonymised, on software metrics and fault presence per module. Except for fault presence, no further issue metrics are provided. PROMISE also has some available data for effort prediction, which is mostly intended for prediction of the effort required to implement an entire project, instead of single issues. Since so little data about the issues are present beyond quantity, we decided not to use this set.

The Eclipse dataset from the University of Saarland[5] [37] contains bug fix information with links to corresponding source code fragments. For every Eclipse module, the fixes that were made there during a certain release are noted. Since the rest of the bug information can be traced to the Eclipse Bugzilla repository, this can be used to do a more low-level analysis of issue impact factors. Similar datasets are provided for AspectJ and Rhino.

As discussed in section 2.3, our work will not use VCS-ITS links, because of the possible bias this introduces[7]. The dataset includes only the linkage between source and issues, without any additional source or issue information. This extra information, such as source metrics or issue status data, has to be looked up separately in the VCS or ITS if required. Because of this, we chose not to include the Eclipse dataset. However, it could be an interesting candidate for a more detailed (e.g. module-level) validation of our results.

The FLOSSMetrics project (see section 2.4) provides a set of tools that is able to extract a Sourceforge issue tracker in an automated way. The database schema used here is very similar to the schema used by Bugzilla. Unfortunately, we were unable to properly run the FLOSSMetrics repository extraction tools. Along with the limited ITS compatibility, this made us decide against using this tool set.

3.3 Data Model

None of the existing tools and data sets we found were adequate for the purposes we had in mind. This made us decide to develop a custom extraction tool, with corresponding data model to store our subject data. This section will explain the data model, while section 3.4 will describe the tool that uses it.

Since we were initially interested in VCSs as well as issue trackers, the data model was designed to include both issue and versioning data. We also chose to include source metrics, because we would like to compare the issue to the source measurement of a project. An overview of the resulting data model is shown in figure 3.3, which will be explained below.

The issues found in an ITS are represented by the three classes `Issue`, `IssueState` and `IssueComment`. Each `Issue` contains only very basic data about submitter and submission date. This is the part of the properties that will not change throughout the issue's lifetime. Most of the issue's properties are stored in a linked list of `IssueState` objects.

[5] `http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/`

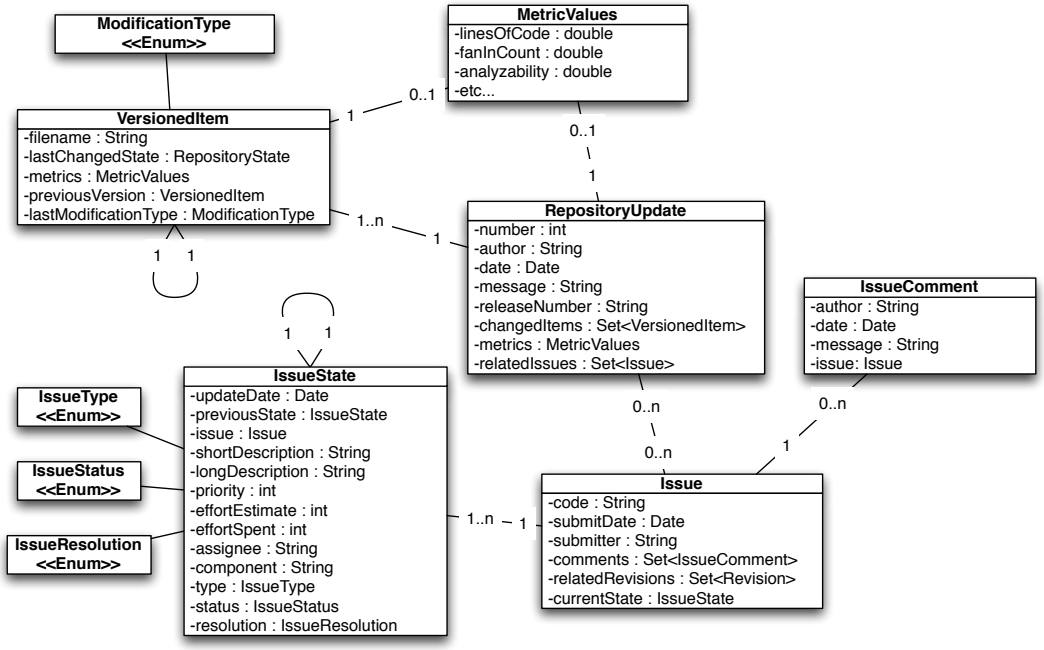

Figure 3.3: Repository database object model

The `IssueState` class depicts the state an issue is in at a given point in time. The properties of such an object indicate the state, priority and other properties of the associated `Issue`. Only one `IssueState` is connected to each `Issue`, representing the current (or last known) state of affairs. An `IssueState` has a previous state, through which it forms a linked list representing the entire history of the issue. The *updateDate* property indicates the date at which the change was made that created the state.

Comparing this model to figure 3.1, it is clear that this approach of storing the history is very different from the common approach. There are two major reasons for this choice.

Firstly, this model makes it easier to determine the repository state at a certain point in time (in the past). To do this, we can simply look at the `IssueState` for each issue that is closest to, but before, the wanted date. Doing this in the common model requires a recursive backwards trace of the changes for each issue, up to the wanted moment.

Secondly, it is possible to find single events in the issue history fairly easily. By this we mean the ability to discover for example when issues where reassigned or when their priority was changed. To do this, we look at the differences between two successive `IssueState` objects. If the wanted field is different, we know that a change event occurred at the *updateDate* of the second state. In the common model, doing something like this would be more difficult. In that case, we can determine when certain changes occurred by looking at the field that was changed. However, reconstructing what the new value was again requires tracing the other changes back to that event.

Instead of requiring us to walk through all changes for an issue each time we want to look something up, our model allows us to do this just once. Our extraction tool converts

the data we gather from ITSs, which is in the common format, to our new model.

We chose a similar approach on the VCS side of the model, where files and other items are stored in the `VersionedItem` table. This is also a linked list of successive versions of an item. This can be linked to a `MetricValues` object, which contains code metrics just for that file. A set of files is contained in a `RepositoryUpdate`, which represents a commit in the VCS. This `RepositoryUpdate` can also be related to a `MetricValues`, which in this case are the metrics for the entire file tree for this revision.

The link between `Issue` and `RepositoryUpdate` is meant to indicate which commits are related to which issues. This functionality is currently unused, because it is not generally possible to reliably determine this linkage. This can only be done for projects which explicitly track such links. For the sake of generalizability and reusability for future work, we chose to include this feature now, without using it.

3.4 Data Gathering

The data model proposed in the previous section is used by a Java tool we developed to extract repository data. Currently, this tool supports the extraction of Bugzilla, Issuezilla, Sourceforge and Jira issue trackers, and the Subversion version tracker. Each of these can be extracted into an XML file, either through the repository itself or through a small shell script we wrote. The tool then reads this file and converts the repository data into our data model, which is stored in a MySQL database. See figure 3.4 for an overview of the workings of our tool.

The approach to extract ITS data differs for each type of tracker. The easiest are Sourceforge and Issuezilla trackers, which both offer a simple way of downloading a repository dump in XML format through a special URL. Bugzilla and Jira are more difficult, since they only offer a list of issues for download. This list only contains basic issue properties without history. In order to obtain the change history, an HTML page has to be retrieved for each issue, and subsequently scraped for the relevant data. We created a few (Bash) shell scripts to provide an easy way to do this. The data gathered here is stored in an XML file with a format similar to that of the other trackers.

The XML files that are extracted from the four supported ITSs have similar data models, as discussed in section 3.3. The Java tool reads these files and converts the data to fit our model. This means it maps the states, resolutions and issue types from the source ITS to the values present in our model. Then, it transforms the change history of all issues from the common model to ours. This is done by walking through the history backwards and creating a linked list of states, one for each change. Afterwards, if multiple changes occur at the same point in time, they are merged into a single state.

To obtain data from a subversion repository, we make use of the subversion command line tools to dump the revision history to an XML file. The command to do this is `svn log --xml -v [repository]`. This XML file is then imported by the tool set and converted to fit the data model.

Finally, we also store the metrics and maintainability appraisals for the system in our model. The SIG SAT is used to calculate these for one or more source code snapshots per

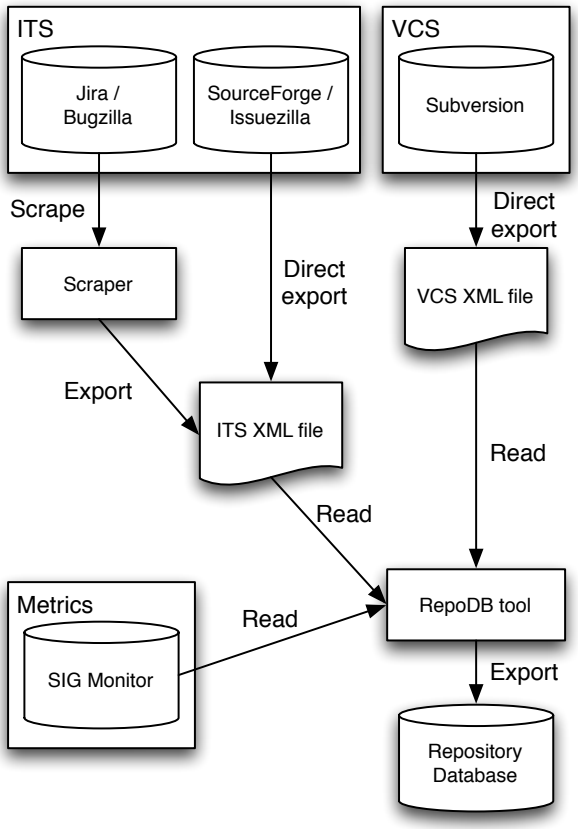

Figure 3.4: Extraction tool data flow

system. The database that is created by this analysis is accessed directly by our tool. The metrics are then read and linked to the VCS revision that represents the snapshot. Snapshots are assigned to the correct revision numbers through a user-supplied configuration file. Since the SAT also supplies metrics at a file-level, it is also possible to retrieve these and attach them to the correct `VersionedItem` object.

Once the data sources are read and the data is converted to fit our model, our tool exports the dataset to a MySQL database conforming to the model. This is done for all three sources separately. In other words, it is possible to first process the ITS and store it in the database, and then do the VCS and metrics later. This means it is also possible to work on partial data sets, i.e. where one of the repositories is not available, if necessary.

The database that is constructed by this tool can be accessed through a Java object model supported by Hibernate. For our further analyses of our data set, we made heavy use of R, an open source statistical package[6]. We created a number of library functions in R that allow us to easily access this database and retrieve the data we need for further analysis. All the analyses in the following chapters make use of these functions.

[6] `http://www.r-project.org/`

Chapter 4

Correlating Maintainability to Basic Issue Metrics

Using the tool we developed in the previous chapter, we can begin to gather the repository data that is necessary for our research. To answer the research questions and determine the relationship between issues and maintainability, we will attempt to find a correlation between the two. This chapter will describe the set-up and results of an experiment we did to find such a correlation.

We start in section 4.1 by describing the objectives and hypotheses we seek to confirm. Section 4.2 describes the subject systems we investigated, whereas section 4.3 describes which measurements we took of those systems. The results are shown in section 4.4 and the conclusions we draw from this experiment are presented in section 4.5.

4.1 Objectives

In order to determine the relationship between software maintainability appraisals and the efficiency and effectiveness of solving issues, we conducted an experiment. The goal of this experiment was to analyse the correlations between software maintainability measurements and issue handling properties. This was done in the context of readily available open source software.

To measure the maintainability of software, we used the SIG maintainability model [21] and corresponding tools. This model is based on the maintainability characteristic of the ISO 9126 software quality standard [23]. It defines four subcharacteristics of software maintainability: analysability, changeability, stability and testability. The SIG has developed a model that assigns systems a rating on these four subcharacteristics based on static source code measurements.

The subcharacteristics of the ISO maintainability model can be interpreted as phases of resolving an issue: Analyse, Change, Stabilize, Test. Since we wanted to determine the impact of the maintainability on issue solving, it would have been ideal if these could be mapped directly to some aspects of the issues. Unfortunately, it is not possible to map this directly to the states an issue passes through (e.g. Bugzilla bug lifecycle, Fig. 3.2). Almost

all the solving effort is concentrated between an issue being *assigned* and *resolved*, with the other issue states not corresponding to any phases in maintenance. It is therefore also impossible to derive a direct mapping between issue properties (e.g. 'time spent in assigned state') and the maintainability aspects. This is why we use aggregate metrics ('mean time to resolution'), in relation to the system property appraisals for each release, to formulate the following hypotheses.

H1 : Higher analysability leads to a smaller mean issue resolution time.

H2 : Higher analysability creates a lower number of issue reassigns.

H3 : Higher changeability means a smaller mean issue resolution time.

H4 : Higher stability leads to less newly submitted issues per LOC.

H5 : Higher stability means there will be less issue reopen events.

H6 : Higher testability means there will be less issue reports per LOC.

The rationales and intuitions behind these hypotheses are as follows.

H1 : Increased analysability means it is easier to locate the cause of problems in corresponding source code, contributing to a quicker solution.

H2 : When it is easier to analyse a system, it should be easier to quickly determine what parts are affected by a problem and who is responsible for those parts.

H3 : If code is easier to change, it is expected that it takes less time to solve issues occurring in that code.

H4 : Stability indicates the tendency of changes to induce new problems. If the stability is higher, we should therefore expect less new problems to occur, leading to less issue reports.

H5 : In software that is more stable, changing things (e.g., solving an issue) causes less problems. This also means issues should less likely be reopened due to unforeseen consequences of a fix.

H6 : More testability means it is easier to test a system. When testability increases, we expect problems to turn up sooner and be fixed sooner in the development cycle. This means that there will be fewer issue reports, since more will be fixed before release.

In order to test these hypotheses, we make use of Spearman's rho to determine correlation between the variables in each hypothesis. We use a one-sided test, since the hypotheses specify the direction of the expected correlation. The independent variables are the analysability, changeability, stability and testability ratings for subject systems. Dependent variables are issue resolution time, number of reassigns, number of issues per LOC and number of reopens.

Name	Version	Date	Period end	Size (LOC)	#Issues
Abiword	2.6.0	2008-03-18	2009-05-06	332132	10941
Ant	1.6.0	2003-12-18	2006-12-13	102028	5192
ArgoUML	0.26	2008-09-27	2009-03-23	166675	5789
Checkstyle	4.0	2005-11-29	2009-07-27	41610	612
Hibernate-core	3.2.0.ga	2006-10-16	2009-08-15	145482	4009
JEdit	4.0	2002-04-12	2003-02-28	54235	3401
Spring-framework	2.5.6-SEC01	2008-10-31	2009-08-10	241913	5966
Subversion	1.5.0	2008-06-18	2009-03-19	218611	3103
Tomcat	6.0.0	2006-10-21	2009-04-21	164715	644
Webkit	6530	2009-02-16	2009-08-10	1255543	22016

Table 4.1: Software versions in dataset

The hypotheses were tested on both a set of snapshots from different projects, one for each project, and for certain projects on a series of snapshots within the same project. This was done to investigate the differences in correlations between projects, i.e. whether the relations discovered vary between projects. Another reason for this approach is to verify the generalisability of the hypotheses.

4.2 Dataset

The data we used for our experiments was obtained from 10 open source projects, see Table 4.1 for an overview. These projects were chosen because they are well-known projects and have publicly available source code and issue repositories. A few of the projects (notably ArgoUML and Checkstyle) were included because of the large amount of previous research done on these projects, e.g. [25, 35, 27].

We made use of the SIG SAT to obtain metrics data and maintainability data for one of the major releases of each project. The SAT provides us with four maintainability subcharacteristic ratings, that we need as independent variables. The subcharacteristic ratings are based on six system properties, which are in turn calculated from static source code measurements [33]. Each subcharacteristic is assigned a rating from 0.5 to 5.5.

To calculate the dependent variables, we used the time frame between the chosen release and the next to determine relevant issue metrics. The rationale here is that the work that was done on issues in this period is influenced by the code properties of that release.

For a specification of the exact properties of an issue we used, as well as the tools used to extract this data, see chapter 3. Table 4.1 shows the version and corresponding date used in our analysis, as well as the date of the end of the release period. It also includes some size indications of the projects.

Because of the presumed distinction between maintainability effects on defects versus other issue types, we chose to initially focus this experiment only on defects. Defects were also thought to be influenced more by product maintainability than tasks, because they

always require an understanding of the source code to solve. Tasks can also be new features that only require addition to the code.

Since invalid issues, like those marked as *duplicate* or *wontfix*, will not be influenced by source maintainability, we left those out. Only issues that have been solved, or might still be solved (i.e. not yet marked as invalid) are included in our dataset.

Some projects should be examined in more detail, to determine the effect of maintainability within a single project, instead of between projects. We expanded the dataset for five of the projects to around 20 snapshots. These projects are Ant, ArgoUML, Checkstyle, Spring and Tomcat. For these projects, source metrics were calculated for each major and minor point release and issue metrics for all intervals between two releases. These intervals were associated to corresponding releases in the same fashion as before. The exact number of snapshots for each project depends on the number of available releases, but all are around 20. These projects were chosen because we encountered them most in literature.

4.3 Basic Issue Metrics

For this experiment, we calculated a number of simple metrics directly from properties of the issues that are related to a release. These metrics are those that were thought to best represent the four maintainability subcharacteristics, leading to the hypotheses discussed above. To recap, the derived issue metrics we calculate from our database to use as dependent variables for each release are the following:

- Mean resolution time of closed defects in release
- Number of defect submissions per LOC
- Number of defect reassignments
- Number of defect reopens

The *mean resolution time* is the mean of all resolution times of defects that are resolved between a release and the next. Resolution time is defined as the time between the moment a defect was submitted and the moment it was marked as being *resolved* and/or *closed*. Sometimes defects are marked as being *resolved* without the defect later being closed. We chose to count these as being finished. If a defect was reopened, we take each open interval to be a separate defect for the purpose of calculating the mean resolution time.

The numbers of defect reassignments and reopens designate the number of times any defect was reassigned or reopened during the relevant time period. Note that this can happen multiple times for a single defect. The number of defects submitted per LOC is the ratio between the number of new defects that are submitted and the LOC size of the system snapshot.

A histogram of the subcharacteristic ratings in the subject systems is displayed in figure 4.1. This shows that our target snapshots have a reasonable spread over the total value range, but there are unfortunately no very good or very bad systems included.

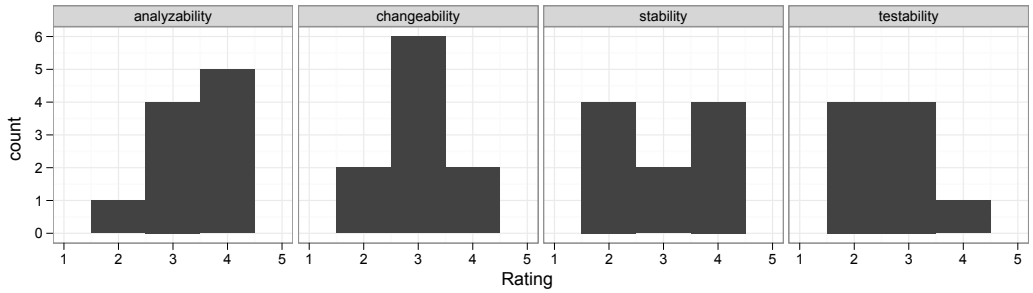

Figure 4.1: Overview of maintainability in subject systems

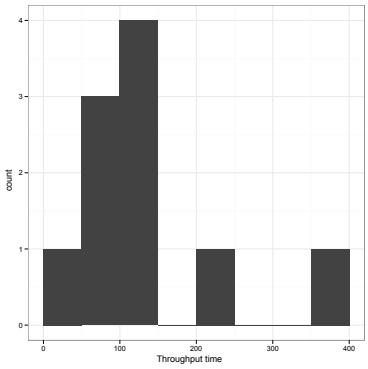

Figure 4.2: Histogram of mean throughput time

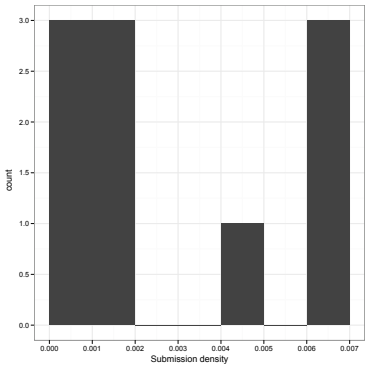

Figure 4.3: Histogram of submissions per LOC

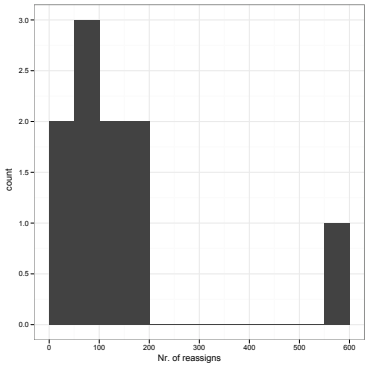

Figure 4.4: Histogram of number of reassigns

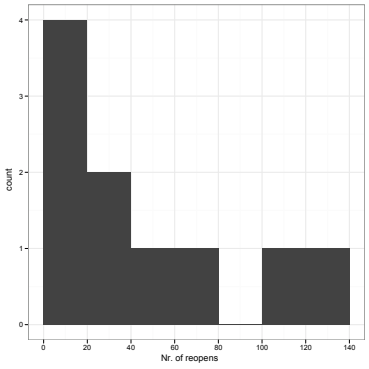

Figure 4.5: Histogram of number of reopens

hypothesis	ρ	p-value
H1	-0.16	0.326
H2	0.19	0.699
H3	-0.21	0.278
H4	0.18	0.687
H5	-0.26	0.234
H6	0.22	0.733

Table 4.2: Test results, all projects

The histograms for the dependent variables are shown in figures 4.2, 4.3, 4.4 and 4.5. As we can see, there is a lot of variability in our data. All distributions look skewed to the left. In the reassignment graph we see an outlier value, which is Webkit at 553 reassignments. Webkit is also by far the largest system in our set, which could explain this. This high value of reassignments does not seem to be caused entirely by size, however. The second-largest system, Abiword, has around half the issues but only 63 reassignments, much less than half that of Webkit. Since this indicates other influences than just size could be present, we did not remove this outlier.

4.4 Correlation Results

To validate the hypotheses, we used one-sided Spearman rank correlation tests between the relevant variables. First, we applied the tests to the single snapshots mentioned in table 4.1. This gives us ten data points to work with. The correlations and p-values for this experiment are shown in table 4.2.

It is clear that there is a severe lack of significance for all hypotheses, indicating that there are too few data points available to reach a conclusion. This result could be improved by using (many) more data points. This would require obtaining issue data and source metrics for many more systems. To find out whether an influence could at least be discovered within projects, the same experiment was also performed on multiple snapshots per system. The results for around 20 snapshots of Ant, ArgoUML, Checkstyle, Tomcat and Spring are displayed in table 4.3. The table again shows correlations and p-values. The stars in the third column for each system indicate which tests are significant at a 95% confidence level. If a dot is shown in this column, the test is significant at a 90% confidence level.

The Checkstyle project shows a few significant correlations, in fact, H2 is quite strongly supported. The two hypotheses dealing with throughput time (H1 and H3) are supported for Checkstyle, and less strongly for Ant. Interestingly, ArgoUML disagrees completely, with correlations directly opposite to our expectations. The extreme p-values for H1 and H3 indicate that there might be a significant correlation opposite to our expectations.

The Tomcat and Spring projects are never near significance and also show weak correlations. Looking at the data, a possible cause for this is that Tomcat contains some release periods that are very short and where no defects were fixed. This reduces the number of

Hypothesis	Ant (N=20) ρ	p-value		ArgoUML (N=20) ρ	p-value		Checkstyle (N=22) ρ	p-value		Spring (N=21) ρ	p-value		Tomcat (N=19) ρ	p-value	
H1	-0.35	0.070	.	0.54	0.993		-0.46	0.019	*	-0.15	0.248		0.21	0.771	
H2	0.07	0.612		0.22	0.821		-0.79	0.000	*	-0.04	0.435		0.70	0.999	
H3	-0.38	0.056	.	0.54	0.993		-0.53	0.007	*	-0.34	0.064	.	-0.24	0.192	
H4	-0.02	0.472		-0.17	0.243		-0.35	0.054	.	-0.12	0.305		-0.36	0.063	.
H5	-0.13	0.289		-0.11	0.324		-0.02	0.468		-0.35	0.059	.	-0.37	0.059	.
H6	0.28	0.885		-0.08	0.367		0.41	0.972		0.11	0.681		-0.01	0.482	

Table 4.3: Test results, individual projects

data points that are usable for hypotheses 1 and 3 further, reducing significance. For Spring, we cannot find an explanation for these results other than having too few data points.

There is considerable disagreement between our projects on the number of reassigns, hypothesis 2. In Checkstyle, this has a strong negative correlation to analysability, supporting our hypothesis. On the other hand, for Tomcat there is an almost equally strong correlation in the other direction. This might be caused by some difference in the issue handling process of these projects that has to do with the process of (re)assigning issues.

Another interesting result from this experiment is the distinction between the first three hypotheses (analysability/changeability) and the last three (stability/testability). The first three seem to get a moderate amount of support, enough to seem promising for further investigation. The last three on the other hand, produce almost no significance and would probably need a significant amount of extra data points to give a conclusive result.

4.5 Discussion

As is evident from the previous section, we are unable to form a strong conclusion about the validity of our hypotheses, due to lack of significance. Only within two of the projects can we draw some conclusions. H1 and H3 get quite strong support within Ant and Checkstyle. Only within the Checkstyle project can we find moderate support for H4 and very strong support for H2. On the other hand, the ArgoUML project shows evidence that in this case, the opposite of H1 and H3 is true.

Since the result from the set of ten different systems is inconclusive, we cannot draw any conclusions about the validity of our hypotheses in general. Only when applied to successive releases within a single project do some maintainability characteristics show a correlation to these issue metrics. In order to reach our goal of comparing systems to another, we will have to develop more generalisable measures. Since both promising hypotheses are based on issue throughput time, we will initially focus on that metric for improvement.

Another possible explanation for the disappointing results is that the issue metrics that are used are very simplistic. In other words, the hypotheses we initially posed are too naive and influenced by many confounding factors. This means we need better approximations for the representation of maintainability in the issue tracker. If we could formulate better intuitions about the exact influence of maintainability on things like issue solving speed or effectiveness, perhaps we can come up with a better issue metric.

A threat to the validity of this experiment is a possible dependency between data points. Since the various data points for a single system represent a series of releases, they are not completely independent. The maintainability ratings of a snapshot, for example, will be influenced by that of the previous snapshot, through the code that is retained from there.

To improve the results of these tests, we could include more projects into our dataset. However, we are restricted by time and effort constraints which make this unattractive. Furthermore, this will only lead to improvements in the inter-project test. The intra-project testing is already making use of a fairly dense distribution of releases for each subject project, sometimes with just two weeks between snapshots. Adding more would increase the influence of the dependency between successive releases, and pose a larger threat to validity.

Considering these results, we decided the relatively simple issue metrics to be an unpromising research direction. To still find an answer to our research questions, we directed our subsequent effort towards finding more detailed approximations of maintainability in the issue tracker. First, we developed two visualisation techniques for the ITS data, to get a better understanding of our dataset. These visualisations will be discussed in chapter 5. Then, we attempted to apply the SIG quality profile model for code metrics to issue metrics, to find a better way to compare the issue handling between systems. This will be discussed further in chapter 6.

Chapter 5

Issue Management Visualisation

The previous chapter used standard statistical tools to analyse basic issue properties. This did not show any significant correlations to software maintainability and we suspect more advanced measures are necessary to be able to reach a conclusion. In this chapter, we will develop some more advanced visualisations of the events in an ITS. The ultimate goal is to use these visualisations to develop an issue-based metric that better captures the notion of maintainability.

Section 5.1 presents the Issue Churn View, a quantitative overview of the changes in the issue tracker. Section 5.2 discusses the Issue Lifecycle View, which is a detailed view of events in the ITS. Two additional case studies are presented in section 5.3, after which section 5.4 concludes the chapter.

5.1 Issue Churn View

The purpose of the Issue Churn View (ICV) is to give an overview of quantitative changes to the issue handling process over time. An ICV shows the state of the ITS per month, in terms of the number of open issues and changes in that number. An example of the graph is shown in figure 5.1, which will be explained in this section.

5.1.1 Definition

For each month, above the time-axis we show in red the number of issues that were submitted in that month. Dark red indicates which ones were also solved in the same month, light red indicates the issues that are left open at the end. The grey part of the bar indicates the issue backlog, i.e. the issues that were left from previous months and haven't been closed this month. Dark grey is the recent backlog (less than half a year) and light grey the long-term backlog (open for more than half a year). Below the time-axis we show the number of solved issues in green. Dark green are issues both submitted and closed in this month, whereas light green are older issues that have been solved in this month.

The reasoning behind this colour scheme is as follows. First, the most important facts that should be derived from this graph are the amounts of incoming and outgoing issues. These are visible in the red and green colours. Note that the category *new and closed*

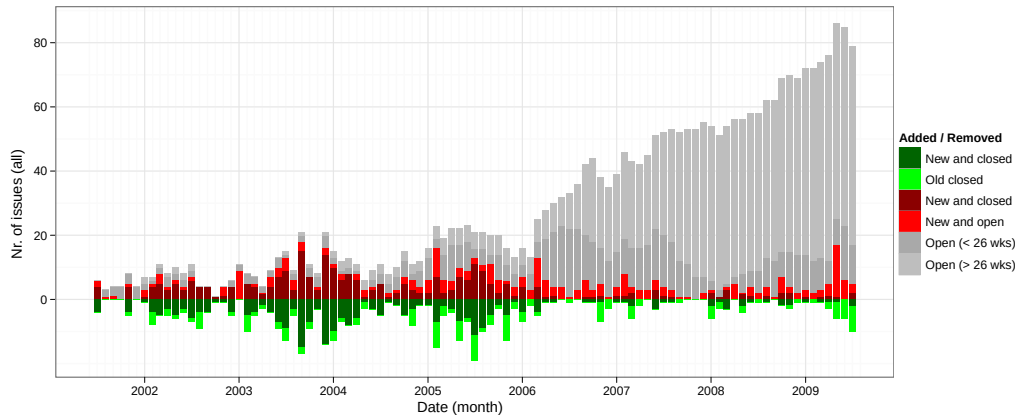

Figure 5.1: Checkstyle issue churn view for defects

is plotted on both sides of the axis, since it contributes to both incoming and outgoing numbers.

Another important aspect of the churn is the development of the backlog, i.e. the issues that could not be solved very quickly. The size of the backlog is shown as the grey part of the graph. Increase of this part should be reason for alarm, since this most likely indicates that issue solving capacity is below what is needed. The reason for splitting the backlog into a light and a dark part is the assumption that issues that have been open for more than half a year are of such low priority that almost no work will be done there. They are out of the immediate scope of the developers. An increase in this part means more and more issues are left unsolved, which could be reason for alarm.

Note that both the red and green bars help in determining the backlog development. The green part shows which part of solved issues are new and which part came from the backlog. This gives an indication of the importance that is placed on reducing backlog. In the red part, we can see which part of the new issues was solved within the month, which can be used as a (crude) measure of efficiency. The light red part is then the part of the new issues that immediately contributes to the increase of the backlog.

An ICV is constructed separately for defects and tasks, since these two categories of issues are presumed to behave differently, in terms of numbers and priority. In the case studies later in this chapter, we will see an example where there is a large behavioural difference that can be clearly seen with this view.

5.1.2 Application

The issue churn view gives us insight into the activity level of the issue solving process. The graph clearly shows the number of incoming and solved issues and can give us an indication of issue solving effectiveness through the display of the issue backlog.

An example issue churn view is show in figure 5.1. This figure shows the churn for the defects in Checkstyle. Immediately, a number of interesting observations can be made

about the Checkstyle defect solving process.

1. From March 2006 until May 2009, both the submission and solving of defects fall back to almost nothing. Since no defects are closed, a relatively large backlog is slowly developing.

2. The last three months of the graph show an increase in activity, both in submissions and resolved issues. This is presumably due to the rising interest in Checkstyle 5.0, which was released in March 2009.

3. Before 2006, there seem to be three distinct phases in the development, where the activity increases for a time, and then decreases again. These phases peak at April 2002, December 2004 and July 2005. This could be evidence of changing community interest, perhaps also due to release cycles. During these phases the solving effort adapts to the changes in incoming defects, so little backlog develops.

As we can see, the ICV gives us a good overview of the quantitative changes in the ITS. We can see changes in activity levels both in issue reporting and issue solving, and we can clearly see how many are left unsolved.

To verify the usefulness of the ICV, we contacted the lead developer of the Checkstyle project, Oliver Burn, via e-mail. We asked him to consider these observations and comment on them. Regarding observation 1, he confirms the lack of activity:

> "The cool down would be to a number of factors, but the most significant would be that the Checkstyle product is now very mature and does not require a lot of development. As such, the interest level of the committers has dropped off as well." (O. Burn, personal communication, December 13, 2009)

For observation 2, the developer agrees with our hypothesis about the cause of the rising interest, but is unable to confirm this completely:

> "Not really sure, I guess putting out the official 5.0 release caused some extra activity."

Regarding observation 3, the developer indicated that these changes were indeed caused by changing community interest. He told us:

> "The three phases of activity you have seen are because of interest in the project. The first is when it was founded, the second when other committers joined the project, and the third when the project was re-architected."

The issue churn view is very suited to the monitoring of system development activity and can be used to track issue solving effectiveness. If, for example, a significant increase in backlog or decrease in the number of solved issues is observed, this can be acted upon. Two more case studies will be discussed in section 5.3 as examples of what the issue churn view can be used for.

5.2 Issue Lifecycle View

As a complement to the ICV, we present the more detailed Issue Lifecycle View (ILV). This view shows the inner workings of an ITS in much more detail, which makes it possible to see some additional information, but loses the ease of use of the ICV.

5.2.1 Definition

The lifecycle view is based on the change history view by Zaidman et al. [35]. The change history view is an overview of the events in a VCS. Simply put, it is a scatter plot of files versus modification dates, coloured based on the type of modification.

For the ILV we do things very similarly. An example which will be explained can be found in figure 5.2. The plot shows the date on the horizontal axis and issues on the vertical axis, sorted by the date they first appeared in the tracker. An issue is represented by a horizontal line segment, which indicates when the issue was marked *open*. On this line are blue dots for comments that are placed on the issue, and yellow dots for *events*. Events are defined as any change in a property for that issue, except for opening, closing or commenting, since those are visible by other means. Examples of events can be reassignment, attaching a patch or changing the priority. Note that a reopened issue will have multiple line segments and that it is not impossible for events to happen while an issue is closed.

The ILV has some correspondence to the churn view. The slope of the leading edge shows the speed with which issues are submitted, something that can also be seen from the churn view. The number of horizontal lines directly above a point in time shows the number of open issues at that point. If necessary, the backlog can be determined by looking at the lengths of the lines. However, it is much less easy to note the number of solves that happen in a time window, especially since the graph often suffers from heavy overplotting.

The advantage of the ILV over the churn view is that it shows the age of issues. This allows us to see the age composition of open issues, something that is only partly possible in the ICV. Using the lifecycle view, we can understand more about how the backlog that is visible in the churn view is being handled. A system with a constant backlog that is not being addressed would have a set of very long lifecycle lines, the backlog, and a set of short lines, issues that are actually being solved. In a system where the backlog is addressed, the line lengths would be more equal, since the older issues are solved quicker, but the younger ones slower.

5.2.2 Application

As we have seen, there are some correspondences between this graph and the ICV. However, because the lifecycle view also shows issue events, it is possible to get additional information about events in the development history. Consider again figure 5.2. We can observe a number of things, some of which are directly related to the ICV.

1. Until 2006, the backlog is very small, only a few issues are still in backlog at this point. From 2006 onwards, the backlog is much larger.

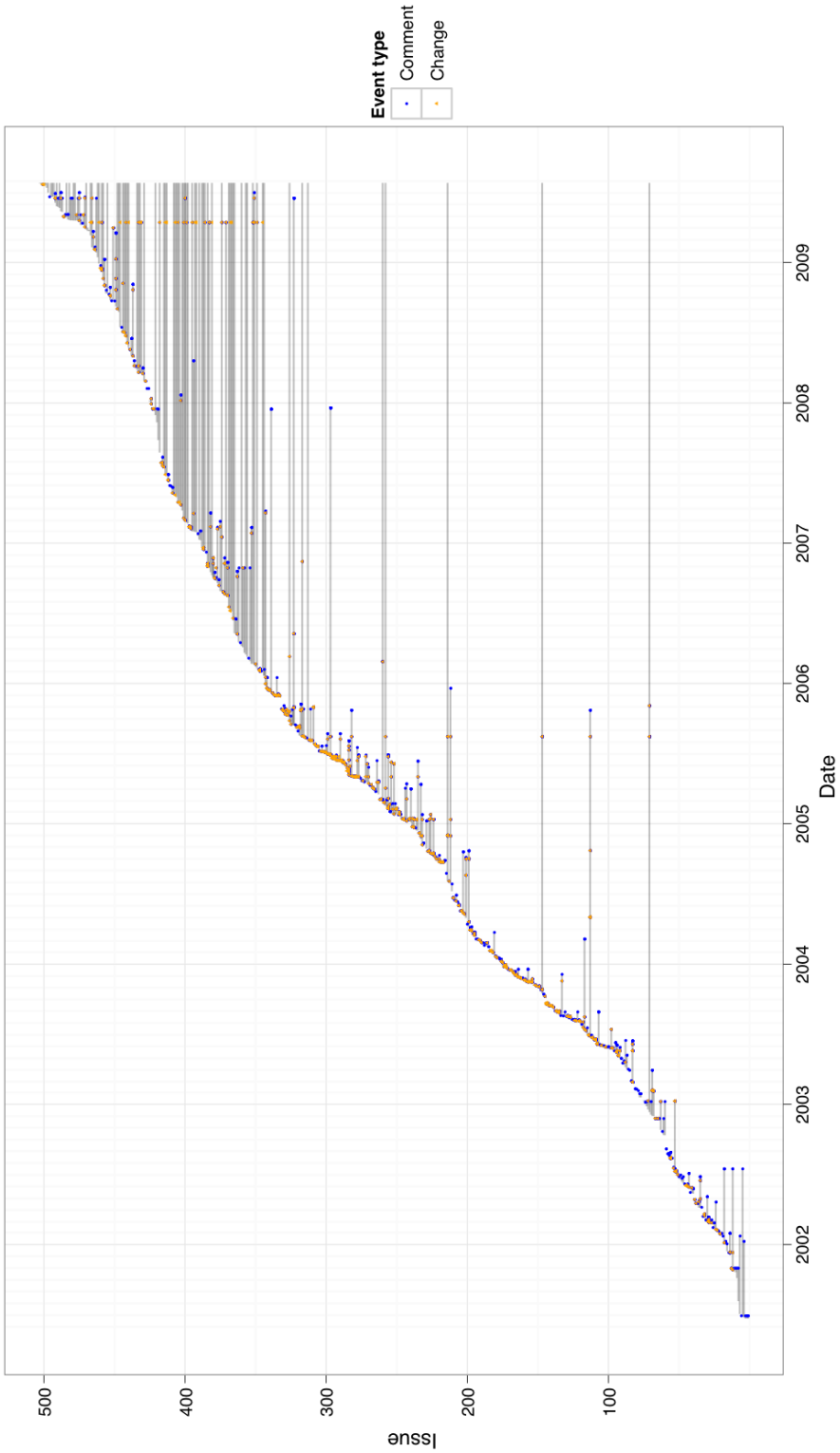

Figure 5.2: Checkstyle issue lifecycle view for defects

2. There are very few events or comments late in the defect lifecycle. Defects are either solved rapidly, or left open for a long time and then suddenly closed, without intermediate action.

3. Around May 2009 there is a vertical line of changes and comments. This could indicate a large clean-up action as work is started on a new release.

The cause of observation 1 is related to the decrease in developer interest, mentioned by the lead developer in the previous section. The maturity of the Checkstyle project causes a decreased interest in further improvement and solving remaining issues.

Observation 2 is likely to be caused by the small number of developers in the Checkstyle project. In our dataset, there are only 4 main committers in the VCS. Therefore, there will likely be little need to discuss issues extensively or change developer assignments or other properties. This is what would lead to the observed small number of comment and change events for each issue.

We verified the third observation in the Checkstyle ITS and found that a large number of issues were reassigned to the main developer on 15-04-2009. This causes the visible yellow line in the ILV. This can hardly be called a clean-up action as we thought. It is more likely caused by development picking up again after a period of inactivity, similar to the peak at the end of the ICV that was discussed by the lead developer.

5.3 Further Case Studies

In the previous two sections we defined the ICV and ILV and showed some example observations that can be made for the Checkstyle project. We have seen that these visualisations allow us to see the changing activity levels in the ITS, as well as the development of a large backlog. The ILV also allows us to see the age of open issues, and clean-up activities that are performed.

To further demonstrate the use of the two ITS visualisations, we will discuss two more projects from our dataset as case studies. These are the two projects that show the most interesting developments in their issue trackers. We found that more than half of the projects in our dataset showed a steadily increasing backlog for both defects and tasks, with a fairly constant amount of solving. The two projects discussed here, Spring and ArgoUML, show more variation and are therefore more interesting to discuss further. Of course, the discovery that most of our subjects have an ever-growing number of open defects is still an alarming discovery.

5.3.1 Spring

The Spring project is a platform for building enterprise Java applications. For this study, we only used the data for the Spring-Framework project, for which the ITS can be found at `http://jira.springframework.org/browse/SPR`.

The defect and task churn views for Spring can be seen in figures 5.3 and 5.4 respectively. From these graphs a number of observations can be made:

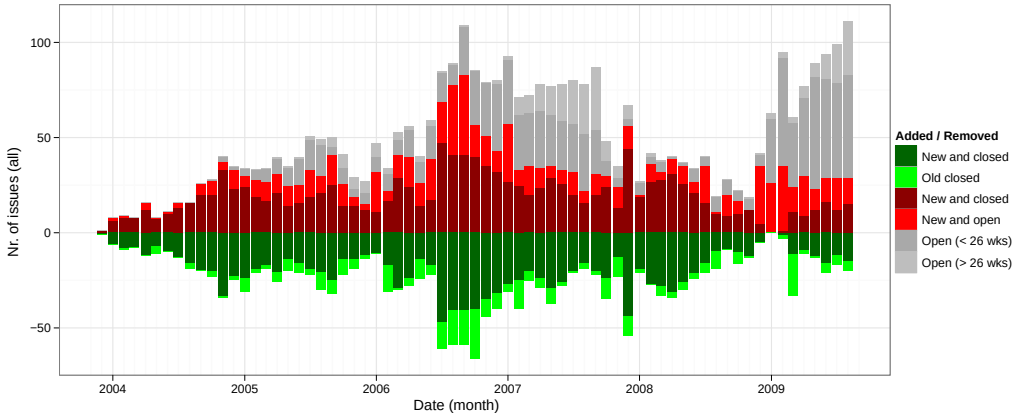

Figure 5.3: Spring issue churn view for defects

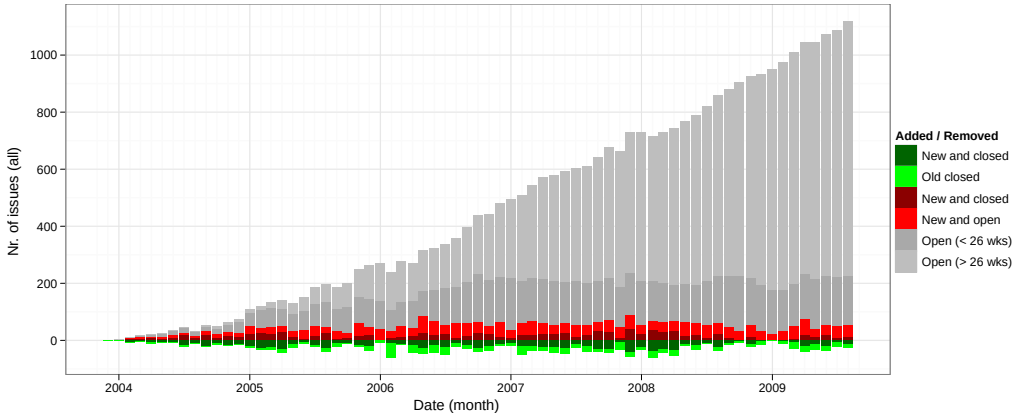

Figure 5.4: Spring issue churn view for tasks

1. There is a large difference in backlog development between the defects and tasks. Apparently, much more emphasis is placed on keeping the number of known defects down, whereas the issues of other types are kept open much longer.

2. The defect development is fairly balanced between incoming and outgoing, but there is a large jump in the number of incoming and resolved defects in June 2006. It takes a few months for this increased activity to go back to normal levels, and more than a year for the backlog to be resolved.

3. There is a strange spike in November 2007. Possibly this is where a new release or a widely publicized problem caused a large inflow of small new defects that were quickly resolved.

4. Defect solving activity seems to stop almost completely around January 2009. It is very strange for development effort to die down like this, we suspect an external cause

for this, such as large changes in the project set-up that caused a lapse in issue solving.

5. At the end of the graph period, the backlog is increasing at a rapid pace. Perhaps the project gained a lot of popularity, or the conventions for defect handling were changed. This could also be a result of what happened in January 2009.

The backlog between January and September 2007 seems to be fairly constant. Could this be because the oldest issues are not addressed, or because they are, but no time is left to solve newer issues, which then take their place? The growing part of long-term backlog seems to point at the former situation. By looking at the ILV, we can answer this question, see figure 5.5.

We see that a few defects from that time period remain open until September 2007. At that point, there appears to be a clean-up event, where they are all closed at the same time. However, the other issues in the time interval do not show much difference with the rest of the graph. The real situation is somewhere between the two options: most issues got resolved fairly quickly, sometimes contributing to the short backlog, but a few stay open for much longer and form the long backlog that is suddenly closed.

The ILV also clearly shows the rapid backlog accumulation in the last few months that we also saw in the churn graph. If we look at the ILV for the tasks, figure 5.6, we see a similar picture to what we saw in the ICV. A large number of issues remain open until the end of the graph interval, in this case leading to a very cluttered graph. Here we can also see clear evidence of similar clean-up events like we saw in the Checkstyle graph, in the form of vertical stripes.

Unfortunately, the lead developer of Spring was not available for the verification of these observations. This means we had to validate the observations for the Spring project ourselves, with publicly available information. Our findings are listed below, numbered according to the observations.

1. The Spring framework project does not list any policies for issue handling on its website or documentation. Therefore, we were unable to confirm why there is such a large difference between the defect and task churn views. In the ITS, we see that a lot of the tasks that are open seem to be feature requests from users. These are not being worked on, but are kept open as reference, instead of being marked *wontfix*.

2. At June 20, 2006 the first release candidate for Spring 2.0 was released, a major release. From the issue data we have available, we can tell that after this date, there is a large amount of defects reported for this release and subsequent release candidates. The final release of 2.0 is at October 3, 2006, which is when the activity starts going down again. Therefore, we believe this sudden jump to be caused by this big release and the accompanying jump in public interest. A lot of these issues are solved very quickly (within days), so the backlog does not increase much and a corresponding peak is visible in the closed defects.

3. Looking at the Spring release history, we can see that Spring framework version 2.5 was released at November 15, 2007. Since this was an important milestone, we believe the spike in defect activity was caused by sudden interest in this release. Indeed,

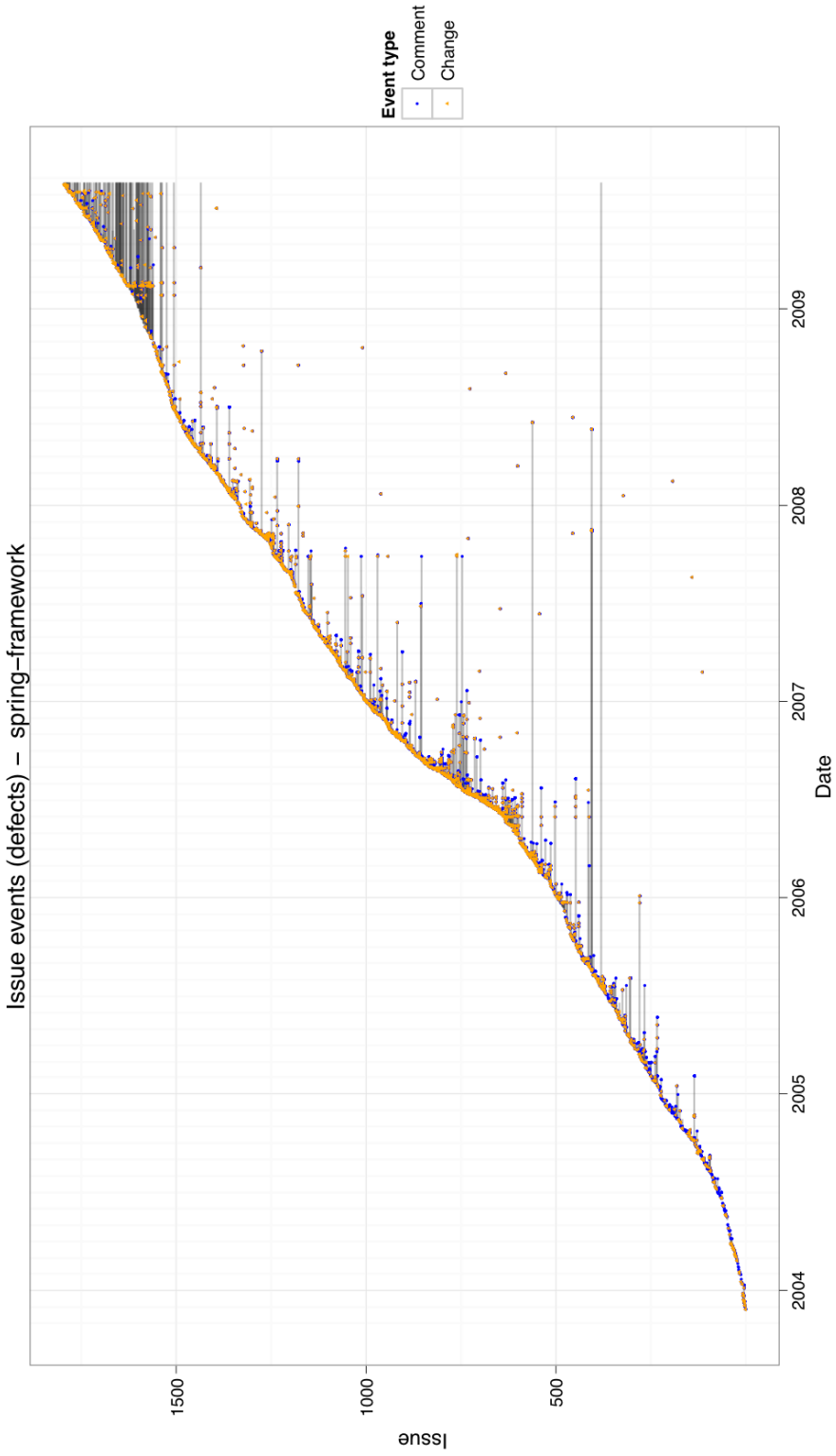

Figure 5.5: Spring issue lifecycle view for defects

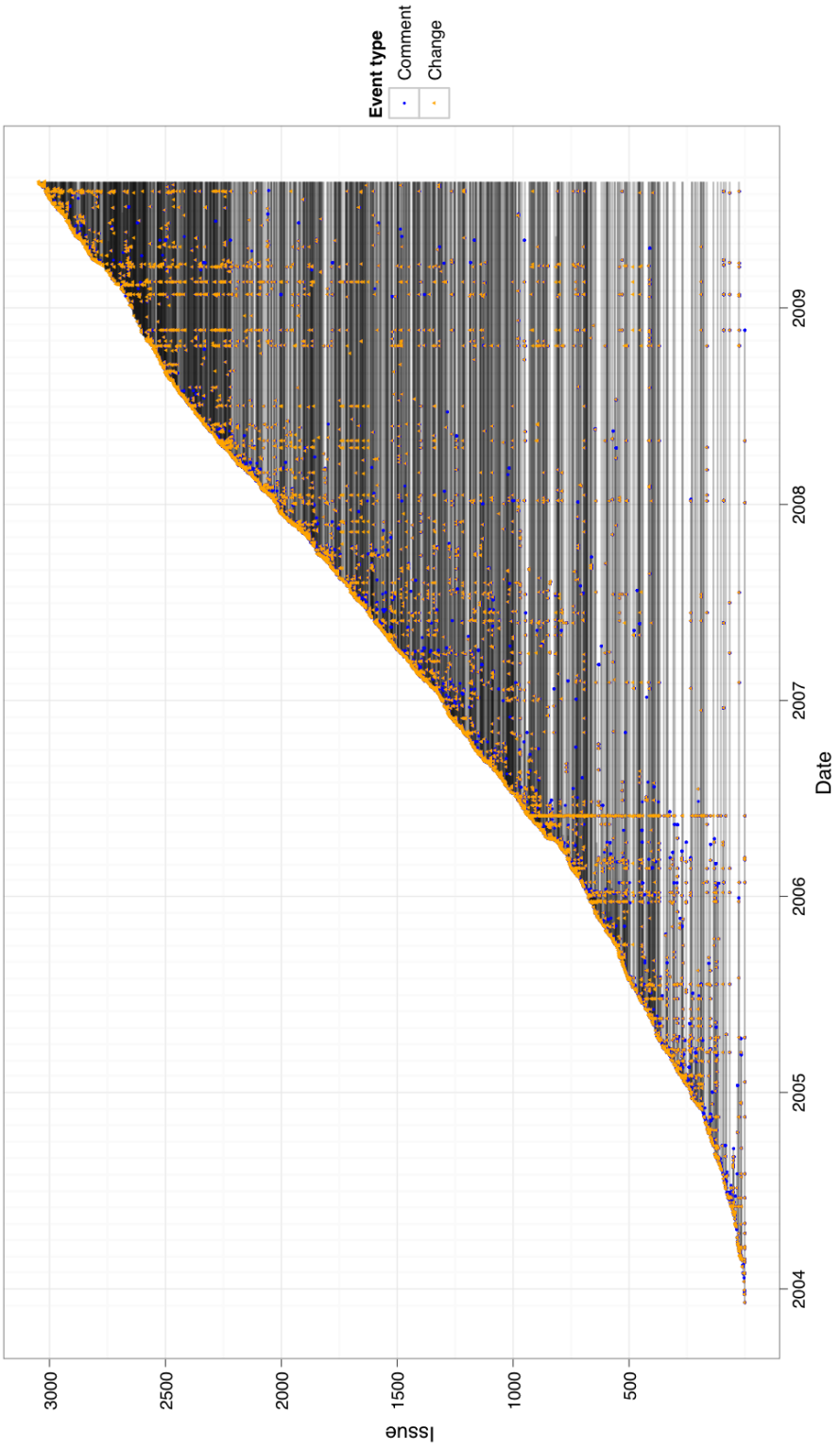

Figure 5.6: Spring issue lifecycle view for tasks

when we investigated the defects that were submitted around this time, we see that the first half of the month was filled with problems in 2.5 release candidates, whereas the second half had only defects in the 2.5 version. There was no release in December 2007, and the defects reported in that month also affected earlier versions instead of just the latest. All of the defects reported in these months are marked as fixed for version 2.5.1, released in January 2008. This is why this spike has no lasting effects on the backlog.

4. According to the Spring website[1] the source repository was moved from Sourceforge CVS to an SVN repository. From the information available in these repositories, we can see that this happened around December 2008. The move placed the Spring 2.5 branch into a maintenance repository, and development for version 3.0 picked up, in a new repository. The first milestone release for version 3.0 was at December 5, 2008.

   While development continued in the new repository, this could be the reason for the lack of defect solving in this period. A large part of the reported defects in this period were for the 2.5 branch, that was now in maintenance. In the ITS, we can see that the large amount of defects fixed in February 2009 was due to the second 3.0 milestone, released February 25, 2009.

5. We were unable to find a direct cause for this increase in defect backlog. It is possible that the development of Spring 3.0 took up too much development time to be able to solve defects as effectively as before. Almost all defects reported here do affect the 3.0 milestones, so this is reasonable.

The manual verification gives us confidence in observations 2, 3 and 4 since those can be directly related to important releases. We also have moderate confidence in the correctness of observation 1, since a prioritisation difference between tasks and defects seems logical. We are still unsure about observation 5, especially since previous large releases did not cause a large backlog growth. Additional information from the developers would definitely improve confidence here.

5.3.2 ArgoUML

ArgoUML is a UML modelling tool that has often been a subject of software engineering research (e.g. [25, 35]). Its issue tracker can be found at `http://argouml.tigris.org/project_bugs.html`. For these graphs, we only regarded the issues related to ArgoUML itself, i.e. the *argouml* component in the ITS, not the website or documentation.

The churn view for ArgoUML's defects and tasks are shown in figures 5.7 and 5.8 respectively. The following observations can be made:

1. There is a short burst of activity in 2000, but the real start seems to be in December 2001 for both defects and tasks. Maybe this indicates a change in development practice, where tasks were unused at first.

[1] `http://www.springsource.org/about`

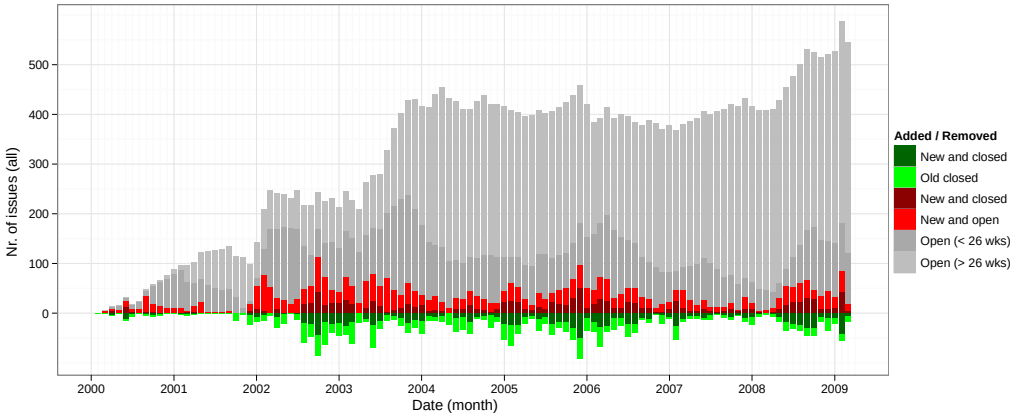

Figure 5.7: ArgoUML issue churn view for defects

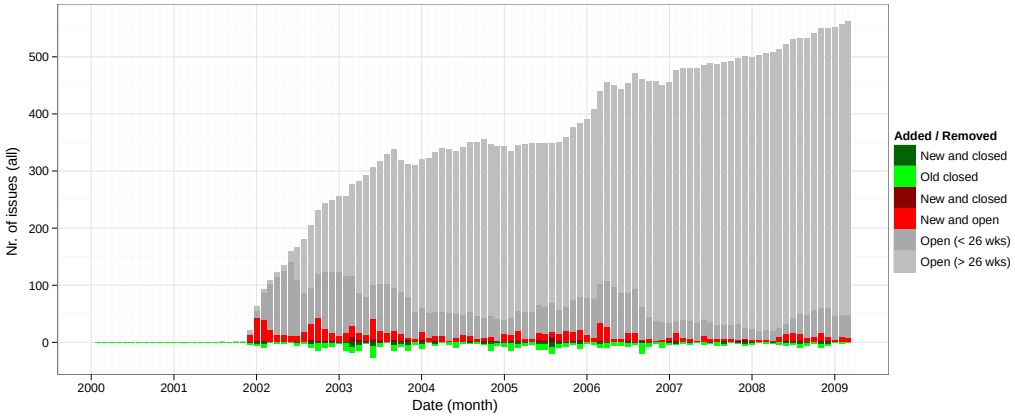

Figure 5.8: ArgoUML issue churn view for tasks

2. The capacity or interest in defect solving seems to drop halfway through 2003, leading to an increased backlog. This is matched at the end of the year by a decrease in incoming defects, stabilizing the backlog.

3. Around November 2005 there is a noticeable increase in the amount of submitted and solved defects, perhaps corresponding to a major release.

4. The defect backlog is fairly constant, with a few sudden jumps over a few months. However, the task backlog is slowly increasing, indicating a lack of capacity or priority.

Looking at the lifecycle views, figures 5.9 and 5.10, we see a similar picture. Defects and tasks develop in the same way, with many remaining open for a long time. Note that there appear to be some events that happen before the corresponding issue has been submitted.

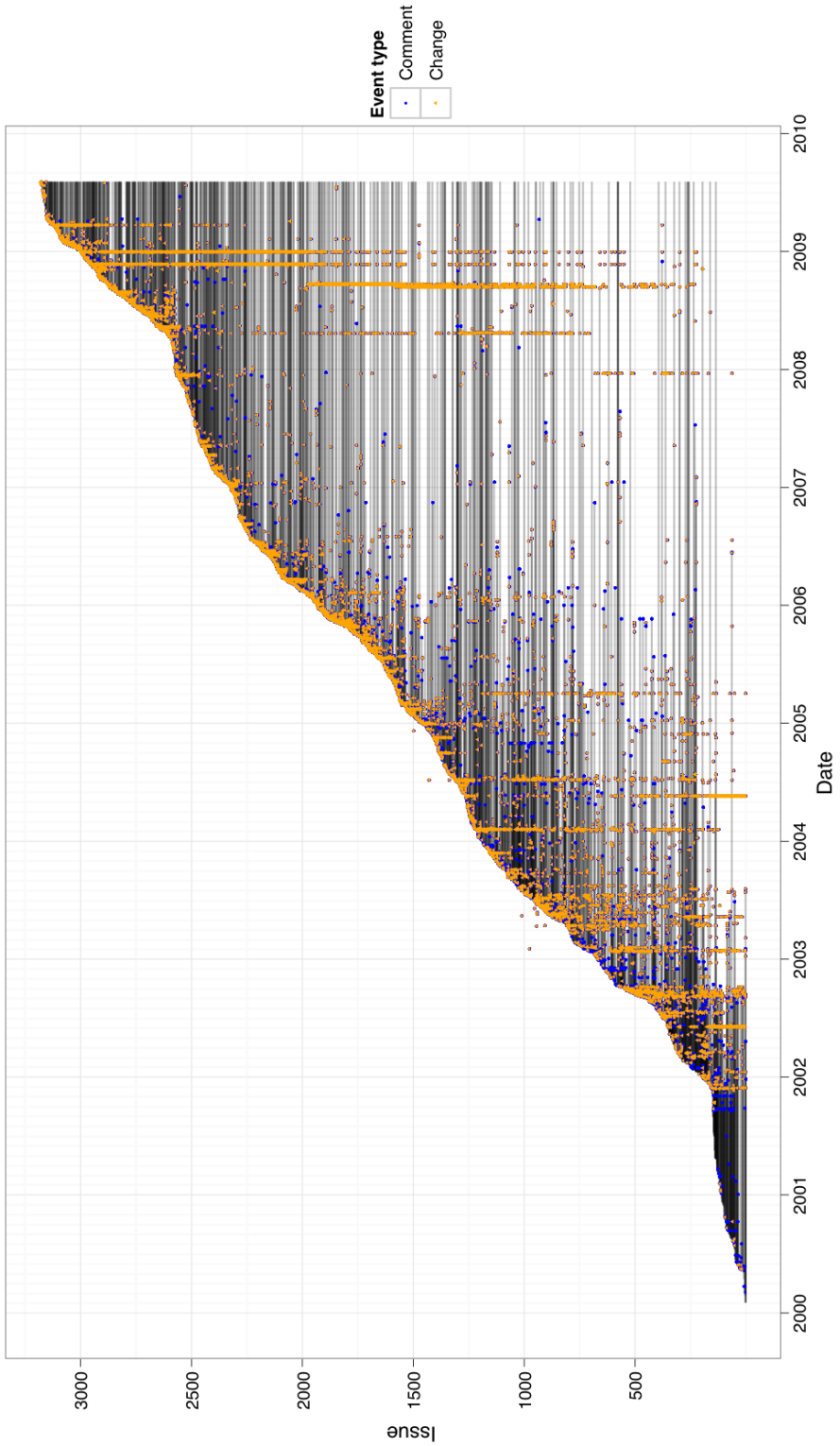

Figure 5.9: ArgoUML issue lifecycle view for defects

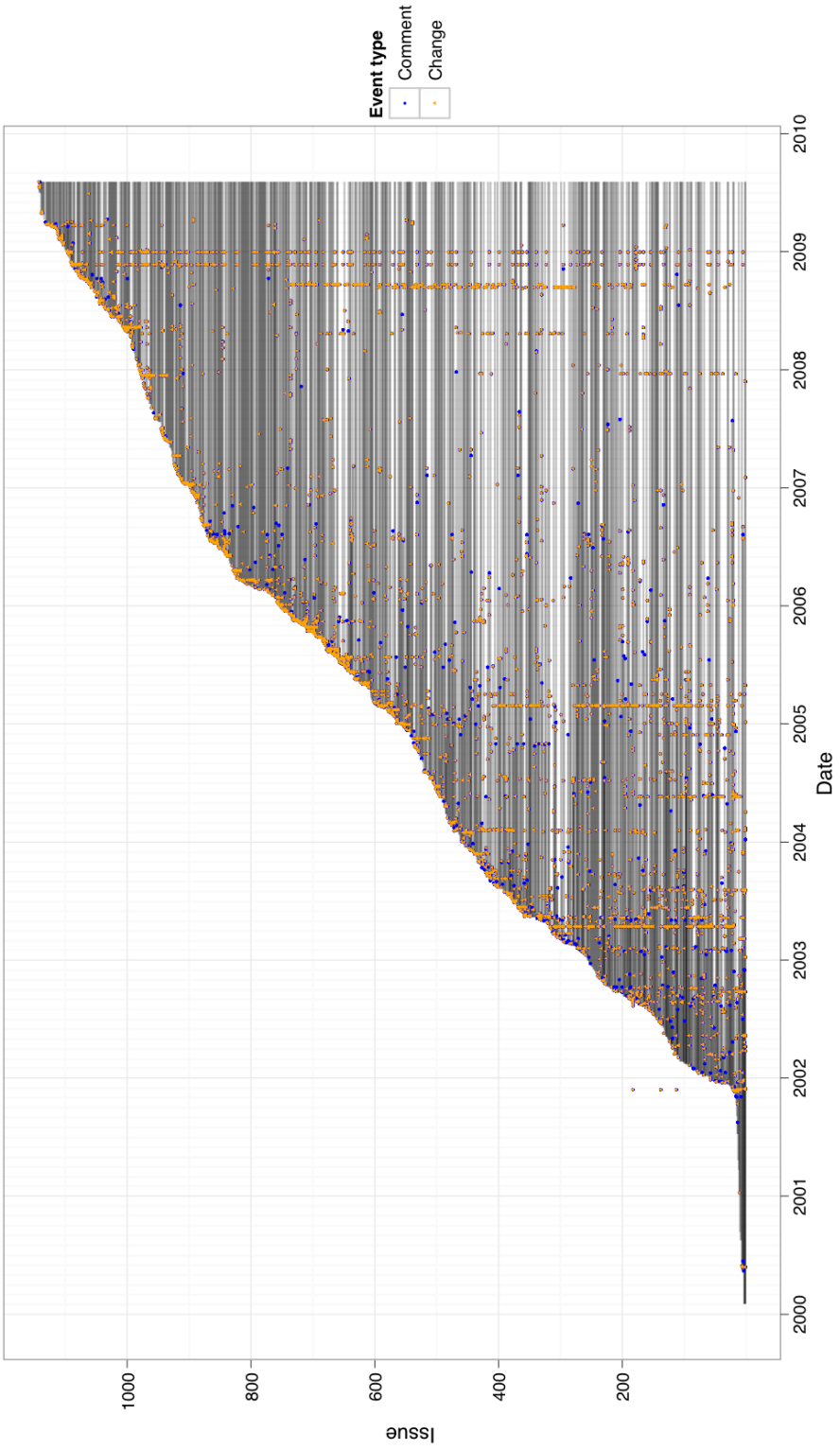

Figure 5.10: ArgoUML issue lifecycle view for tasks

These anomalies are also present in the original ArgoUML issue tracker. The presence of these events has no influence on any of our other analyses.

5. These two graphs show an interesting phenomenon that was also slightly visible in Spring and Checkstyle: large simultaneous actions. Both graphs have clearly visible vertical lines of changes and comments. Perhaps these are caused by changing the priority of a large number of old open issues, or assigning them to a different target release.

We suspect these are the result of tracker-wide change or clean-up operations, where a large number of issues was changed. This could for instance happen if issues are marked to be solved in a new release, or moved to a different release.

Again, we contacted the project lead, in this case Linus Tolke, for verification purposes. Regarding observation 1, he was not quite sure on the cause of this, especially considering this was long ago. ArgoUML went through a number of changes in the issue handling process. At first, tasks were used, but later on, they were all changed to defects so only one type was used. This would explain the absence of tasks for the first part of the graph. Later on, tasks were used again (L. Tolke, personal communication, December 11, 2009).

The second and third observations were attributed to changing project popularity and interest. Especially the period around November 2005 was a busy period, just before the alpha for ArgoUML 0.20. According to the project lead:

> "My guess is that this is because of a change in development effort and project popularity. [...] Looking at old announcements we can see that November 2005 is when we entered the alpha period for the 0.20 release. This was a period of a lot of alpha and beta releases. I guess that could mean that there was a lot of activity around that [time]."

Regarding observation 4, we suspected the number of tasks was growing more than the defects due to a difference in process or priority. The project lead confirm a difference in priority, though he does not provide a complete explanation for the difference:

> "Yes, the defects of priority P1 and P2 are considered blockers for releases so there is a difference."

Finally, observation 5 was attributed to the verification of fixes. ArgoUML officially requires fixes for issues to be verified by other developers than the fixer, using the *verified* issue state. Unfortunately, this is an underused feature, so the lead sometimes cleans up the issues that have been open for a time:

> "Nobody bothered to verify issues. Since [2008], I have been verifying and closing issues a certain amount of time after the stable release instead starting with the oldest issues i.e. the ones included in the oldest releases."

5.4 Discussion

This chapter showed two issue tracker visualisations that we developed, the issue churn and lifecycle views. By applying these views to three example projects, we demonstrated their ability to visualise non-trivial aspects of the issue handling process. We verified a number of observations we made and found that all interesting events in the graphs indeed correspond to events in project history.

Looking back at the case studies, we can distinguish a few typical trends in issue handling that these visualisations are capable of bringing up.

Activity : All three ICVs showed some form of changing development activity, especially in the defect graphs. This is visualised through decreasing numbers of solved issues.

Phased popularity : Checkstyle and ArgoUML showed clear evidence of changing project popularity in the ICV. This is visualised by phases of increasing activity, followed by a decrease, causing a peak in the graph. Again, this is especially visible in the defect graphs.

Major releases : In the ICV, very large releases create spikes or sudden increases in incoming defects. All three projects show changes due to busy release periods.

Prioritisation : ArgoUML and Spring show a difference in handling between defects and tasks in their ICV.

Clean-up actions : The ILV can clearly display the large clean-up actions that are sometimes performed in an ITS. All three case studies show the vertical stripes that indicate this.

Backlog development : Both ICV and ILV show the backlog development trend, which is increasing in almost all of our cases. This is especially alarming in the ILV, causing the graph to be very cluttered.

As a final remark, we suspect the change in development practice visible at the start of ArgoUML is also something that could be visible in more projects. We believe these graphs are valuable tools to investigate a project's issue handling process. Further case studies will undoubtedly reveal more patterns that are common between projects.

The visualisations developed in this chapter give us insight into the issue handling process of a single project. They only allow a limited form of qualitative comparisons between projects. In the next chapter, we will develop a metric that allows us to compare issue handling between projects.

Chapter 6

Quantification of Issue Management

The previous chapter focused on providing a qualitative visualisation of an ITS. We would also like to provide a more quantitative measure of the correspondence between software maintainability and what goes on in the ITS. Inspired by the large differences in issue handling times displayed by the visualisation of the previous chapter (and also in section 4.3), we decide to target the issue throughput time with this.

We apply the SIG quality profile method to the issue throughput times as recorded in the tracker. This provides us with a comparative measure of issue resolution times for a system snapshot. This *throughput rating* is then compared to the maintainability ratings obtained through the SIG rating engine to determine whether there is any relation between the two. We again apply Spearman's rank correlation for this.

Section 6.1 will explain what part of the dataset was used to construct this measure, as well as how exactly we measure throughput time. Then, section 6.2 will explain the quality profile model and how we applied it. In section 6.3 we will examine how the rating we constructed relates to software maintainability ratings. Finally, section 6.4 discusses the results.

6.1 Definitions

The measure we developed for issue handling, called the *throughput rating* is based on the time it takes for issues to be resolved. In other words, we look at the time an issue is marked as being in the *new* or *assigned* state in the issue tracker. We take this time to be a rough approximation of the actual time it takes to solve the issue, and via that the difficulty of solving it. Since this is directly influenced by the source code maintainability, we expect there to be a correlation between throughput and maintainability ratings.

We define Issue Throughput Time (ITT) as the time an issue is in an open state, i.e. not *closed* and/or *resolved*. If an issue has been closed and then reopened, all open intervals count towards the ITT, but the intervals where the issue was closed do not. We take this as a very rough estimate of the effort that was spent on solving the issue, for lack of better data available.

Similar to what we did in chapter 4, we order our full issue dataset by versions (snapshots) of the available systems. The source code metrics and maintainability ratings are also available in that form, allowing us to have both the throughput and maintainability ratings for each snapshot.

For each system version, we count as relevant issues those that are closed and/or resolved between that version and the next. We assume that most, if not all, of the work on solving an issue will be performed just before it is closed. Because of this, we can use the date and time at which an issue was closed to determine to which software version it belongs. To ensure each version has a minimum amount of issues associated to it, we remove all snapshots that have 5 or fewer issues. This makes the issue set for each version less likely to be influenced by single outlier issues.

The throughput rating for a version is then calculated based on the issues belonging to it. The maintainability ratings for the system are again provided by the SIG SAT. Our hypothesis is that there will be a positive correlation between both ratings. In other words, issues will be resolved quicker in a system that is more maintainable.

We make a distinction between issues of type *defect* and other types, indicated as *tasks*. These were discovered to be handled differently in the previous chapter, leading us to believe they are impacted by maintainability in a different way. The rest of this chapter will only deal with defects. Because defects inherently have a higher priority, we suspect that they will be impacted most by the maintainability. Again, we also removed all duplicate or otherwise invalid defects from the dataset, since they (by definition) did not result in work on the source code.

6.2 Quality Profiles

The SIG quality profile model [21] is designed to distinguish systems relatively to each other based on a metric that has a *long tailed* distribution. In the case of defects, this is done by primarily looking at the defects that take a long time to solve. The assumption is that the defects that can be solved quickly are relatively easy in any system, whereas the hard defects will be influenced the most by differences between systems.

We will explain the working of the quality profile model while we apply it to our dataset. The quality profile model subdivides the ITT values into four risk categories. These are called *risk* categories because they represent the risk a defect in each category poses to the system.

The categories are constructed by determining the three threshold values that determine in which category an issue is placed. These values should be chosen such that their discriminative power is maximal. To ensure this, we look at a percentile plot of the dataset, figure 6.1.

The black line in the figure represents the throughput percentiles for all defects in our dataset. On the background in light colour are the percentile plots for each snapshot individually. A point on the graph shows what the ITT value is that a certain percentage of defects is below. In other words, if the point $(40, 30)$ is on the graph, 40% of defects for that system will be resolved in less than 30 days.

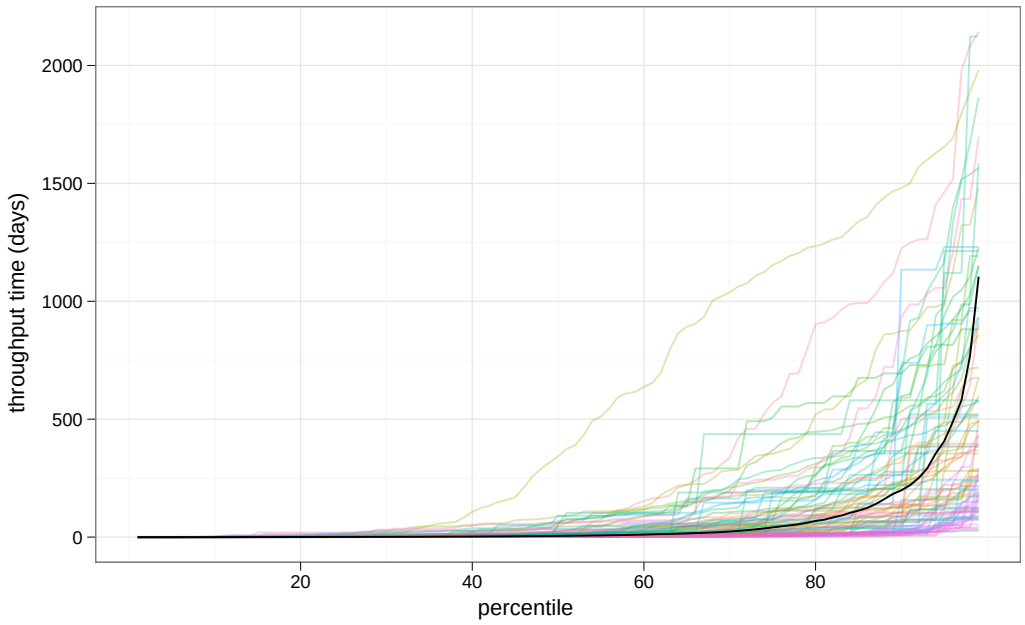

Figure 6.1: System percentiles for issue throughput

Category	Thresholds (days)
Low	0 - 23.6
Moderate	23.6 - 68.2
High	68.2 - 198
Very high	198 +

Table 6.1: Threshold values for ITT

The point here is to take three points along this graph that show significant variation between the systems, so the resulting quality profile will have maximum discriminative power. Here, we use the 70th, 80th and 90th percentiles of our data set as threshold values for the *moderate*, *high* and *very high* categories respectively. This translates into the threshold values displayed in table 6.1. We can see that our model considers defects that are solved within about three weeks to be low-risk. The very-high risk defects on the other hand, could be open for over half a year.

These threshold values allow us to create a *risk profile* for each snapshot. A risk profile is simply a subdivision of all defects in the dataset into their corresponding categories, based on their ITT. Figure 6.2 shows these profiles for our snapshots. Already, we can see some differences emerging between projects. It seems the Spring snapshots have much less high risk issues than for example the ArgoUML snapshots.

To put a number on these differences, the quality profile model assigns each snapshot a

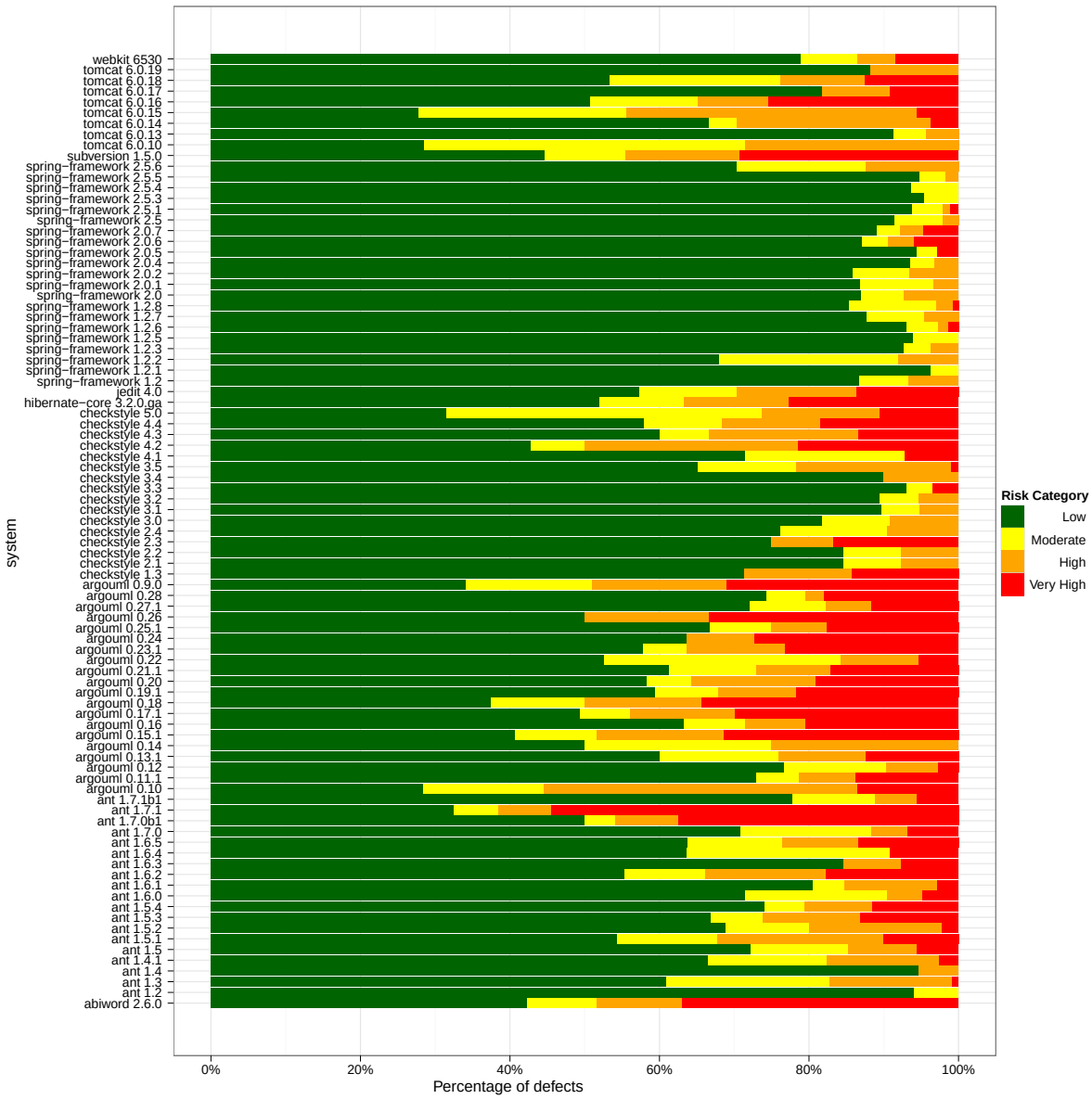

Figure 6.2: Throughput risk profiles

Rating	Moderate	High	Very High
*****	7%	0%	0%
****	25%	25%	2%
***	43%	39%	13%
**	43%	42%	35%

Table 6.2: Threshold profile for ITT

rating from 1 to 5 stars. This rating is based on the percentages of issues that are in each risk category. Each rating level allows only a certain percentage of the defects to be in each of the risk categories. In order to get more granularity in the ratings, the star value is interpolated to yield a number in the range $[0.5, 5.5]$. The calibration of the profiles is such that 5% of the systems will receive a 5-star rating, 30% four stars, 30% three, 30% two and the last 5% one star.

The SIG quality profile tools provide an automated way to perform this calibration. Applied to our data, the rating profiles are as shown in table 6.2. The values in this table represent the maximum percentage of defects that may be in a category to still get the corresponding rating. For instance, to get a four-star rating, at most 25% of defects can be moderate risk, 25% high risk and at most 2% very high risk. This profile shows a large emphasis on differences in the highest category, where only small percentages are allowed. Issues in this category are assumed to be influenced the most by differences between snapshots, as said at the start of this section.

6.3 Throughput and Maintainability Ratings

Applying the threshold profile we developed in the previous section to the risk profiles from figure 6.2 gives us the throughput ratings. This rating now tells us how quickly defects were resolved for a certain system snapshot, relative to other snapshots in our dataset. The ratings for our data are plotted in figure 6.3, grouped per system and plotted versus the snapshot's date.

Of course, the interesting aspect of this rating is whether it is correlated to the maintainability rating of the systems under test. We test this using a Spearman rank correlation test of the throughput rating versus the SIG maintainability rating. As before, a one-sided test is used, because we expect these correlations to be positive. We also include the four maintainability subcharacteristics and six system properties as defined by the SIG model [33]. The results of this test are shown in table 6.3. All correlations, except for unit interfacing, are significant at a 99% confidence level.

The first six rows of the table are the system properties. These are measured directly from the system source code by the SIG tools. As we can see, most of these correlate positively with the throughput rating. The correlations to volume and duplication are low, which is surprising. We expect a larger system to potentially be impacted by a defect in more places, and it will take longer to find the proper fixing location in such a system. An

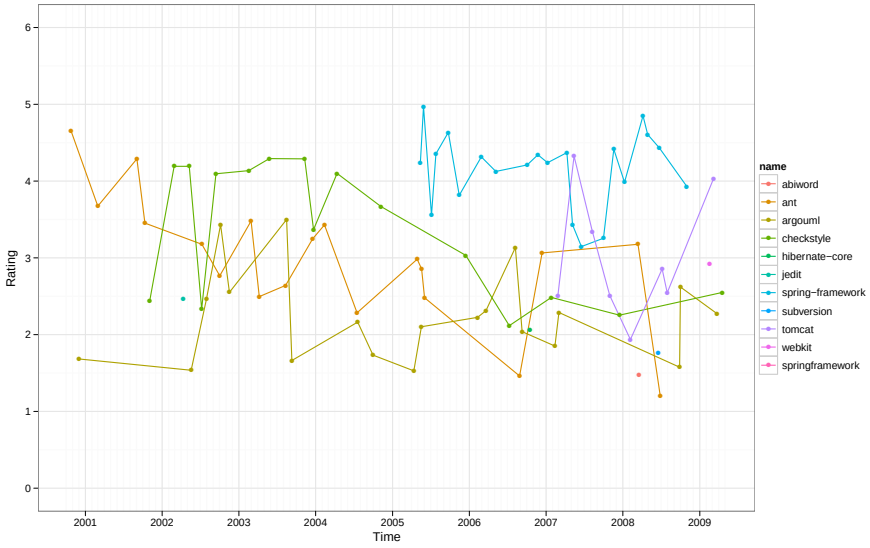

Figure 6.3: Throughput ratings over time

Level	Throughput vs.	ρ	p-value
System properties	Volume	0.31	0.002
	Duplication	0.30	0.002
	Unit size	0.50	0.000
	Unit complexity	0.48	0.000
	Unit interfacing	-0.16	0.931
	Module coupling	0.49	0.000
Subcharacteristics	Analysability	0.52	0.000
	Changeability	0.63	0.000
	Stability	0.39	0.000
	Testability	0.51	0.000
Total	Maintainability	0.61	0.000

Table 6.3: Correlation between throughput and maintainability ratings

	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling
Analysability	X	X	X			
Changeability		X		X		X
Stability					X	X
Testability			X	X		

Table 6.4: Influences between system properties and maintainability characteristics

increase in duplication should also increase ITT, because a fix might need to be performed in more places at once. On the other hand, the low correlation could be due to fixes perhaps only impacting small parts of the systems at once, so size does not matter much.

Unit complexity and size are measures that have to do with internal complexity and size of the parts of the system, rather than the whole. We expected these to be positively correlated to the throughput rating, since we suspect increased unit complexity to cause the code to be more difficult to understand. This would intuitively cause defects to take longer to resolve. For unit size, the same reasoning goes as for volume: larger units are more difficult to understand and modify. The correlations we found are fairly good, supporting our hypotheses.

Unit interfacing and module coupling are related to the complexity of the connections between software units (e.g. classes). It is quite surprising that these are in such disagreement. We would expect more complex interactions inside the system to have a large impact on how easy it is to understand what is going on and solve defects quickly. This is primarily measured by module coupling, which is derived from the number of calls between modules. The results support our expectations in this case. Unit interfacing is derived from the number of parameters software units have, i.e. the complexity of its interfaces. This does not seem to have anything to do with solving defects quickly. Unfortunately, the result is not significant, so more research is needed to investigate this.

From the six system properties, the SIG calculates five-star ratings for the four quality subcharacteristics as defined by the ISO 9126 standard [23]. These ratings are calculated according to the matrix in table 6.4 [33]. This shows which maintainability characteristics are thought to be influenced by which system properties.

Comparing this matrix to our correlations, we can see that the stronger correlations are amplified by this aggregation step. Analysability and changeability are quite strongly correlated to the throughput rating. Intuitively, this make sense for the throughput times. Software that is more analysable is easier to understand. That means it is easier to understand what the cause of a problem is, and where the solution should be located in the source code. Software that is more changeable will be easier to modify correctly. Both aspects will cause the ITT to go down.

Stability is influenced by the interfacing metrics which are in disagreement. This causes its correlation to be lower as well, though significant. Intuitively, this could be explainable. Stability measures the ability to change the system without causing unwanted side-effects or later problems. In terms of defects, we suspect this will lead to more defects being submitted later on, rather than causing the current defects to take longer to resolve.

Testability measures the ease with which the system can be (unit) tested and is based on unit volume and complexity. Since these two have good correlation to the throughput rating, testability does as well. However, not all of the system versions we analysed included test code. This lead us to believe that there might be significant differences in how the testing of defect fixes is done between systems. The validity of this correlation is therefore debatable. A dataset with systems where the teting process is better known is needed to strengthen this result.

The final maintainability rating is constructed by averaging the four subcharacteristic ratings. Again, we see that the highly correlated parts overrule the lower one. It seems that maintainability is definitely correlated to the speed with which defects are fixed.

6.4 Discussion

In this chapter, we have constructed a metric that allows us to compare the issue handling process between system snapshots. The metric is based on the issue throughput time. We have seen that this metric has a positive correlation to the software maintainability as calculated by SIG tools.

This result indicates that in a software project that possesses higher code maintainability, defects will be resolved quicker. However, the correlations we discovered are not extremely high. We suspect this is caused by the numerous confounding factors that are still present in the data. Given the coarse level of detail of this study however, these results are very encouraging.

One of the confounding factors is the time an issue is open, without anyone working on it. Currently, we take the entire time an issue is open as throughput time. Not all this time is spent actually working on the issue. As said, we assume the work is carried out in a single block, at the end of the issue life time. However, we are unsure on how much work is performed exactly. We expect that the amount of actual work done will correlate even better to the maintainability measures. A more detailed dataset is needed to investigate this, because in the ITSs for the systems we investigated, the actual resolution effort cannot be determined.

Another factor is the level of detail at which this study was performed. By looking at the system-level, we ignore the internal differences in systems. Defects in complex parts of the system will probably take longer to resolve than those in less complicated parts. Unfortunately, we are unable to determine which issues affect which parts of the systems, so such a detailed analysis is not possible for our dataset. Again, a dataset that provides more detail could improve results.

We have now performed this analysis only on defects, because those were assumed to be influenced most by maintainability. It would be interesting to repeat the experiment for

tasks. Our expectation is that the correlations and significance levels will be lower, so this might be paired with expanding the dataset to prevent inconclusiveness.

A change that could be made to improve the explainability of the rating we built is to adjust the threshold values for the risk categories. These thresholds could be rounded to 3 weeks, 10 weeks and half a year. This does not significantly change the result, but improves the explainability of the model enormously. From the point of view of a software project, solving a percentage of issues within three weeks will sound better than within 23.6 days.

For now, we can conclude that there seems to be a positive correlation between maintainability in general and the issue throughput rating. This could be evidence for a causal relationship, but further research is necessary to confirm such. Especially the software analysability and changeability seem to influence the speed with which issue are resolved. As a recommendation to projects wanting to reduce issue handling effort, we would advise primarily reducing the unit size and complexity, as wel as the module coupling of their code base.

Chapter 7

Conclusions

This chapter concludes the thesis. Section 7.1 contains a summary, section 7.2 draws conclusions. In section 7.3 we discuss a number of threats to the validity of this work and in section 7.4 we give suggestions for future extensions to this work.

7.1 Summary

We have developed a Java tool and corresponding data model that allows us to extract and store issue repository data. Such data was obtained for ten open source projects. Along with the issue data, we obtained source code metrics for the projects using the SIG Software Analysis Toolkit (SAT).

We investigated basic metrics derived from the ITS and attempted to find a correlation between these and the maintainability of the corresponding project. We found significant correlations for issue throughput time in the Checkstyle project. This metric also seemed promising for Ant and Spring. However, when tested over all projects, we were unable to confirm our hypotheses. To remedy this situation, we decided to apply the SIG quality profile model to our data, specifically focussed on Issue Throughput Time (ITT).

To get a better understanding of what goes on in an ITS, and how the ITT develops, we defined two visualisation techniques. The Issue Churn View (ICV) shows the changes in number of open issues, along with the development of the backlog and solving efforts. The Issue Lifecycle View (ILV) shows a more detailed view, of the lifecycles of each issue and the changes that are made to it. The two visualisations are closely related to each other and have been shown to be useful in making observations on the issue handling process.

The SIG quality profile model was applied to the ITTs in our dataset. This provided us with a metric that can be used to compare the issue handling process between different systems, or versions of the same system. The *throughput rating* we developed compares the systems in our dataset relative to each other, rather than on an absolute scale. We showed this rating to correlate positively to software maintainability characteristics. Based on our results, there is reason to believe that software unit size and complexity, as well as module coupling are important factors in the defect solving speed.

7.2 Discussion

In the introduction to this thesis, we formulated two main research questions, to which we will provide the answers here.

RQ1 : Can we distinguish differences in issue metrics between different projects or project phases?

The issue visualisations we developed show us the qualitative changes in both the amount of issues and the issue lifecycles. Using these plots, we can distinguish clear differences in the way issues are handled between different projects, for example through the way the backlog develops.

The plots also show evidence of the changing conditions during a single project's history. The ICV shows us that issue activity can increase around busy release periods and reduce when there is little development activity. Evidence of issue clean-up activities related to project releases can be found in the ILV.

To determine quantitative differences, we looked at some basic ITS metrics for a number of system snapshots, such as number of reassigns and number of newly submitted issues. While these metrics show differences between systems, we cannot explain these from a maintainability point of view. The throughput rating we developed does allow us to observe explainable differences between systems, in terms of issue handling.

Using the quality profile technique, we can see that the differences manifest most clearly in the issues that take longest to solve. The time needed to solve such issues varies much more between projects than the time needed to solve the easier issues. Later, we showed evidence that this could be explained by differences in maintainability between systems.

RQ2 : How does production code maintainability affect the issues reported for a project?

We investigated the relationship between basic issue tracker metrics and the SIG maintainability ratings. We found that we cannot draw a general conclusion based on these metrics. However, we did find a significant correlation between the average throughput time and analysability and changeability for the history of two projects.

Building on this, we defined the throughput rating using the SIG quality profile model. This allowed us to compare systems relatively, instead of on an absolute scale, improving the results. We used this technique to compare systems in terms of defect resolution times.

Based on this experiment, we found that the most significant code properties that influence defects are unit complexity, unit size and module coupling. The most important maintainability characteristics are analysability and changeability. An improvement in those is likely to serve towards a reduction in defect resolution times.

7.3 Threats to Validity

There are a number of thing that could threaten the validity of various parts of this work. Some have been mentioned before, but we present a list here for completeness.

Lack of issue detail : In chapter 3 we noted that we cannot distinguish between different phases in solving an issue, because the data to do this is not available. We use the entire issue lifetime as a metric to compare to maintainability characteristics. Though the results seem good, if we could distinguish between the phases, such as analysis and actual solving, the results could be different. This is a threat to the construct validity of both the basic issue metrics and the throughput ratings. Since we are able to see when issues were (re)opened and closed, we can partially deal with this threat in the case of reopened issues. We do this by filtering out the periods were the issue was closed.

Data point dependency : The internal validity of the experiment done in chapter 4 could be threatened by dependencies between data points. We perform a correlation analysis on successive releases of a single system. These releases are probably not independent of each other, since the maintainability of one release will always be dependent on that of the previous, because they will share (much) source code.

Bias in quality profiles : The dataset for the throughput quality profiles in chapter 6 contains systems with just one snapshot, and systems with multiple. This threat to internal validity could bias the results towards those systems that are represented most. We do not believe this is the case, since the ratings vary quite a lot between snapshots, see figure 6.3. This means there will be no large clusters of points that influence the correlations.

Possible lack of generalisability : All data we used in this thesis is derived from open source software that uses an ITS, and all but three of the software versions we used are written in Java. This constitutes a threat to the external validity of our work. Since commercial software products are generally developed in a different, more centralised fashion from open source project, it is possible that our results will not hold for commercial projects. There is also the possibility that other languages than Java favour different styles of issue handling, which could change the results. Lastly, we believe that the use of an ITS promotes a certain rigour in handling issues. The maintainability effects could be different when a more ad-hoc development style is used.

Confounding effects : We did not measure some aspects of software projects that could have an influence on the issue handling process, such as team size or project popularity. Such factors could lead to larger or smaller numbers of issues being reported or fixed. Using the visualisations from chapter 5, we saw that such things are in fact visible in the ITS. This constitutes a threat to the internal validity of our work. However, since we believe this will not influence issue resolution times much, and these are our main focus, we believe the influence of these factors to be small.

7.4 Future work

There are a number of ways in which this work can be improved and built upon. We present a few of our ideas below.

Throughput ratings for tasks : We have applied the SIG quality profile model only to the defects in our dataset. It would be interesting to see whether it is also applicable to tasks. We suspect it is, but that the measured correlations are different, because tasks are solved in a different way from defects, e.g. with lower priority.

Expand the dataset : The dataset we used consists of ten open source systems, four of which are represented by multiple versions. Increasing the size of this set will help in strengthening the confidence in the throughput rating results. It will be especially interesting to add commercial systems and see whether these behave differently from open source systems.

Normalize the throughput ratings : It is clear there are still a number of confounding factors present in the throughput ratings. For example, differences between team size between projects, that influence the throughput times. On the other hand, it seems that the correlation between throughput and maintainability ratings holds even better for successive versions of a single system. We suspect that a number of confounding factors could be eliminated by normalizing the ratings in some way. Again, this would improve confidence in the influence of maintainability.

Lower-level analysis : Ideally, the analysis we performed should be done on the level of source code modules. Knowing where an issue was fixed in the source code is necessary to do this, however. If subject systems that provide such information are available, this is a very interesting research direction. We expect this will give a much clearer picture, because the metrics of irrelevant parts of the system do not influence the result.

Bibliography

[1] Aversano, L., Cerulo, L., and Grosso, C. D. (2007). Learning from bug-introducing changes to prevent fault prone code. *IWPSE '07: Proc. 9th Int. Workshop on Principles of Software Evolution*, pages 19–26.

[2] Ayari, K., Meshkinfam, P., Antoniol, G., and Penta, M. D. (2007). Threats on building models from CVS and Bugzilla repositories: the Mozilla case study. *CASCON '07: Proc. 2007 Conf. of the Center for Advanced Studies on Collaborative Research*, pages 215–228.

[3] Ball, T., Kim, J., Porter, A., and Siy, H. (1997). If your version control system could talk. *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*.

[4] Benlarbi, S., Emam, K. E., Goel, N., and Rai, S. (2000). Thresholds for object-oriented measures. *ISSRE '00: Proc. 11th Int. Symposium on Software Reliability Engineering*, page 24.

[5] Bernstein, P. and Dayal, U. (1994). An overview of repository technology. *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 705–705.

[6] Beyer, D. and Noack, A. (2005). Clustering software artifacts based on frequent common changes. *IWPC '05: Proc. 13th Int. Workshop on Program Comprehension*, pages 259–268.

[7] Bird, C., Bachmann, A., Aune, E., and Duffy, J. (2009). Fair and balanced?: bias in bug-fix datasets. *ESEC/FSE '09: Proc. 7th European Software Engineering Conference and ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 121–130.

[8] Boetticher, G., Menzies, T., and Ostrand, T. (2007). PROMISE repository of empirical software engineering data. *http://promisedata.org/*.

[9] Boogerd, C. and Moonen, L. (2009). Evaluating the relation between coding standard violations and faults within and across software versions. *MSR '09: Proc. 6th Int. Working Conf. on Mining Software Repositories*, pages 41–50.

[10] Briand, L., Wüst, J., Daly, J., and Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems & Software*, 51(3):245–273.

[11] Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[12] D'Ambros, M., Lanza, M., and Pinzger, M. (2007). "a bug's life" visualizing a bug database. *IEEE VISSOFT 2007: Proc. 4th IEEE Int. Workshop on Visualizing Software for Understanding and Analysis*, pages 113–120.

[13] Eick, S., Graves, T., Karr, A., Marron, J., and Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12.

[14] Fenton, N. and Neil, M. (1999a). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689.

[15] Fenton, N. and Neil, M. (1999b). Software metrics: successes, failures and new directions. *Journal of Systems & Software*, 47(2-3):149–157.

[16] Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. *ICSM '03: Proc. Int. Conf. on Software Maintenance 2003*, pages 23–32.

[17] Gall, H., Hajek, K., and Jazayeri, M. (1998). Detection of logical coupling based on product release history. *ICSM '98: Proc. Int. Conf. on Software Maintenance 1998*, pages 190–198.

[18] Gall, H., Jazayeri, M., and Krajewski, J. (2003). CVS release history data for detecting logical couplings. *IWPSE '03: Proc. 6th Int. Workshop on Principles of Software Evolution*, pages 13–23.

[19] Graves, T., Karr, A., Marron, J., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661.

[20] Gyimothy, T., Ferenc, R., and Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910.

[21] Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. *QUATIC '07: Proc. 6th Int. Conf. on the Quality of Information and Communications Technology*, pages 30–39.

[22] Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. *ASE '07: Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 34–43.

[23] ISO/IEC 9126-1:2001 (2001). Software engineering - product quality - part 1: Quality model. *ISO, Geneva, Switzerland*, pages 1–32.

[24] Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131.

[25] Kim, S., Pan, K., and Jr, E. W. (2006a). Memories of bug fixes. *FSE '06: Proc.14th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 35–45.

[26] Kim, S., Zimmermann, T., Pan, K., and Jr, E. W. (2006b). Automatic identification of bug-introducing changes. *ASE '06: Proc. 21st IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 81–90.

[27] Lubsen, Z., Zaidman, A., and Pinzger, M. (2009). Using association rules to study the co-evolution of production & test code. *MSR '09: Proc. 6th Int. Working Conf. on Mining Software Repositories*, pages 151–154.

[28] McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

[29] Mockus, A. and Votta, L. (2000). Identifying reasons for software changes using historic databases. *ICSM '00: Proc. Int. Conf. on Software Maintenance 2000*, pages 120–130.

[30] Ratzinger, J., Pinzger, M., and Gall, H. (2007). EQ-Mine: Predicting short-term defects for software evolution. *Lecture Notes in Compure Science*, 4422:12.

[31] Sandusky, R., Gasser, L., and Ripoche, G. (2004). Bug report networks: Varieties, strategies, and impacts in a F/OSS development community. *MSR '04: Proc. 1st Int. Workshop on Mining Software Repositories*, pages 80–84.

[32] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? *MSR '05: Proc. 2005 Int. Workshop on Mining Software Repositories*, pages 1–5.

[33] Visser, J. (2009). SIG/TÜViT evaluation criteria trusted product maintainability. *Software Improvement Group*.

[34] Xu, J., Ho, D., and Capretz, L. (2008). An empirical validation of object-oriented design metrics for fault prediction. *Journal of Computer Science*, 4(7):571–577.

[35] Zaidman, A., Rompaey, B., Demeyer, S., and van Deursen, A. (2008). Mining software repositories to study co-evolution of production & test code. *ICST '08: Proc. 1st Int. Conf. on Software Testing, Verification, and Validation*, pages 220–229.

[36] Zimmermann, T., Nagappan, N., Gall, H., and Giger, E. (2009). Cross-project defect prediction. *ESEC/FSE '09: Proc. 7th European Software Engineering Conference and ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 91–100.

[37] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. *PROMISE '07: Proc. 3rd Int. Workshop on Predictor Models in Software Engineering*, page 9.