

MASTER'S THESIS
MASTER OF SCIENCE IN SOFTWARE ENGINEERING
UNIVERSITY OF AMSTERDAM

Indicators of Issue Handling Efficiency

and their Relation to Software Maintainability

Dennis Bijlsma

University of Amsterdam
E-mail: mail@dennisbijlsma.com

Supervisors

prof. dr. Paul Klint	University of Amsterdam
ir. Miguel Alexandre Ferreira	Software Improvement Group
dr. ir. Joost Visser	Software Improvement Group

AUGUST 24, 2010

MASTER'S THESIS
MASTER OF SCIENCE IN SOFTWARE ENGINEERING
UNIVERSITY OF AMSTERDAM

**Indicators of Issue Handling Efficiency
and their Relation to Software Maintainability**

Dennis Bijlsma

Abstract

The Software Improvement Group has developed a model that assigns a rating to the maintainability characteristics of a software system, based on the ISO/IEC 9126 standard. In theory, the ratings in the SIG Quality Model should have a number of consequences in practice. For example, an increase in maintainability should lead to more efficient issue handling.

We present four indicators that can be used to measure the efficiency of issue handling. These are *issue resolution time* (the total time an issue is in an open state), *enhancement ratio* (the percentage of resolved issues that are of type enhancement), *project productivity* (resolved issues per LOC), and *developer productivity* (resolved issues per LOC per developer).

To determine the relation between software maintainability and issue handling, we perform an empirical study that uses the source code and Issue Tracking System data from a number of open source projects. We find a positive correlation between all four indicators of issue handling efficiency and software maintainability as measured by the SIG Quality Model.

Keywords Software maintainability, source code metrics, software process, issue tracker mining, productivity.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering, at the University of Amsterdam. The work that led to this thesis was done under the supervision of prof. dr. Paul Klint, ir. Miguel Alexandre Ferreira, and dr. ir. Joost Visser, at the Software Improvement Group.

I would like to start by thanking my direct supervisor at the SIG, Miguel Ferreira, for helping me with setting up the experiments, finding a direction in which to continue my research, answering lots of questions, and being the most thorough proofreader in the history of mankind. I am of course also grateful to my other supervisors, Joost Visser and Paul Klint, for their valuable feedback that gave insight in how you should write a document like this.

I would also like to express my thanks to the people who proofread this thesis. Apart from my supervisors these are Frank Versnel, Arseni Storojev, and Sarina Petri. Special thanks go out to Frank, who was writing his thesis at the SIG in the same period as me. Being able to reflect with a fellow student was very helpful.

Also thanks to everyone at the SIG for a pleasant working environment, where people are intelligent, helpful, and open to questions. Thanks to the researchers Tiago Alves, Jose Pedro Correia, Xander Schrijen, and Eric Bouwers for answering most of these questions.

Finally, I would like to thank my friends, for caring beyond the routinely asking how my thesis is progressing.

*Dennis Bijlsma
Amsterdam, the Netherlands
August 2010*

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Research questions	3
1.3	Research method	4
1.4	Definitions	5
1.5	Outline	6
2	Background information	7
2.1	Issue Tracking Systems	7
2.2	Issue life cycle	8
2.3	The Software Improvement Group	9
2.4	ISO/IEC 9126	9
2.5	The SIG Quality Model	10
2.6	Related work	11
3	Preceding experiment	15
3.1	Unified model for ITS data	15
3.2	ITS Data extraction tool	16
3.3	Issue backlog	17
3.4	Issue resolution time	17
3.5	Issue resolution time rating	17
4	Issue resolution time	19
4.1	Criteria for issue resolution time	19
4.2	The dataset	20

4.3	Resolution time ratings	22
4.4	Correlation results for defects	23
4.5	Correlation results for enhancements	24
4.6	Quantification of issue resolution times	25
4.7	Discussion	27
4.8	Conclusion	28
5	Extending the dataset	30
5.1	Criteria for systems	30
5.2	The new dataset	31
6	Enhancement ratio	33
6.1	Results	34
6.2	Discussion	37
6.3	Conclusion	38
7	Project and developer productivity	39
7.1	Determining the number of developers	40
7.2	Results	41
7.3	Discussion	43
7.4	Conclusion	44
8	Conclusion	46
8.1	Discussion	48
8.2	Threats to validity	48
8.3	Future work	51
	Bibliography	57

Chapter 1

Introduction

Unlike most forms of maintenance, software maintenance is not aimed at keeping the object that is maintained in good condition. Instead, it is aimed at modifying the software for changing user needs, a changed environment in which it operates, or to correct faults, after the software becomes operational [7]. IEEE standard 14764 [16] defines software maintenance as the following:

“Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.”

Although software maintenance is an activity that only starts after delivery, several surveys have indicated that maintenance typically accounts for 70 to 80 percent of a system’s costs [23,1,3]. Thus, creating maintainable software can make software development significantly more cost-effective [36,33].

Because of its importance, software maintainability is considered one of the six characteristics of software quality by the ISO/IEC 9126 international standard for software product quality [17]. This standard splits maintainability into four sub-characteristics: analyzability, changeability, stability, and testability.

The Software Improvement Group (SIG) has developed the SIG Quality Model [13,31], a model for measuring software maintainability based on the sub-characteristics defined in ISO/IEC 9126. This model assigns a rating to each of the sub-characteristics, and to maintainability itself. The ratings are determined by mapping a number of source code metrics to the different sub-characteristics.

The SIG uses this model to help its clients reduce maintenance costs and to increase the technical quality of their software [21]. Although software maintenance only starts after delivery, the model can already be used during construction to get an indication of the system's maintainability.

Since having highly maintainable software reduces maintenance costs, the effects of having a particular rating in the SIG Quality Model are noticeable in practice.

Software systems that are used in a real world environment must change in order to remain useful [22]. Requests for change are usually referred to as *issues*. An issue can be a bug in the software, a request for enhancement, or a task that needs to be performed on request. The term *issue handling* is used to refer to the process of reporting and solving issues.

In professional software development it is common [15, 10] to use an Issue Tracking System (ITS). An ITS is used as central place to store all issues. It can be used internally by members of the development team, but it can also be a public website that is accessible by all users of the system.

An ITS provides valuable information. This information can be used to form an opinion on the state of the software, to plan future versions, to divide work between developers, and to inform users about progress.

The data stored in the ITS can also be used to form an opinion about a project's issue handling efficiency. Issue handling is efficient if issues are generally solved within a short time period. It is interesting to look at the influence software maintainability has on this efficiency. In theory, having highly maintainable software will have a positive influence on issue handling efficiency.

1.1 Motivation

We expect that having highly maintainable software leads to more efficient issue handling. This relation between software maintainability and issue handling efficiency can be direct, but it is also possible that an external factor (for example, attention to quality) influences both software maintainability and issue handling efficiency.

If the results of our experiments indicate that there is indeed a relation between software maintainability and issue handling efficiency, it will encourage developers to write software of higher technical quality. It will also prove that the effects of having a certain rating in the SIG Quality Model are noticeable in practice.

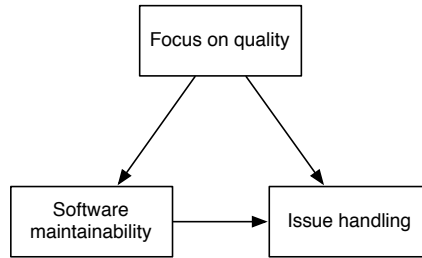


Figure 1.1: Expected relations between issue handling and software maintainability.

To determine if the relation exists, a number of indicators for issue handling efficiency needs to be defined. We can then determine if there is a correlation between these indicators and the SIG Quality Model, which is an indicator of software maintainability.

A previous study [25] has already defined one indicator for issue handling efficiency, *defect resolution time*, and found a correlation between the indicator and the SIG Quality Model ratings. Apart from extending this indicator to other types of issue, it will be useful to define more indicators of issue handling efficiency that can be used to measure this efficiency from different perspectives.

Apart from using them to determine the relation to software maintainability, the indicators for issue handling efficiency can be used to compare this efficiency between projects.

1.2 Research questions

The past sections have listed a number of points that should be further researched. They lead to the following questions that should be answered by this thesis:

[RQ1] Does the correlation between defect resolution time and the SIG Quality Model ratings still exist?

[RQ1.1] How much shorter or longer are defect resolution times for each of the SIG maintainability ratings?

[RQ2] Does the correlation also exist for issues of type *enhancement*? And if so, is it stronger?

[RQ2.1] How much shorter or longer are enhancement resolution times for each of the SIG maintainability ratings?

[RQ3] What other indicators of issue handling efficiency can be defined?

[RQ3.1] Are these indicators correlated with the SIQ Quality Model ratings?

[RQ3.2] What are their values for each of the SIG maintainability ratings?

1.3 Research method

The relation between issue handling and technical quality will be observed by means of an empirical study. The experiments in this study will be performed on a dataset that consists of a number of open source projects.

The projects in the dataset are selected based on size, used programming language, and number of issues in the respective ITS. A number of snapshots will be taken for each project, although this number will not necessarily be the same for each project. A snapshot consists of the source code and ITS data taken at a specific date. These snapshots are created using a tool that extracts ITS data, the source code, and the source code's change history. This tool (further explained in Section 3.2) saves data from different ITSs in a unified model, so that it can be compared. Because of the way ITSs are being used, it is not certain that all issues are a good indication of the project's issue handling practice. An attempt is made to remove all issues that do not follow typical ITS usage from the dataset. The criteria for "typical ITS usage" will be defined later in this thesis.

Part of this study is based on a previous study by Bart Luijten of the Delft University of Technology in 2009 [26, 25]. Luijten found a positive correlation between the SIG Quality Model ratings for system snapshots and *defect resolution time*, the time defects for a snapshot were in an open state.

Initially, a replication of Luijten's experiments is performed to determine if his conclusions are still valid if we exclude issues that do not conform to typical ITS usage. After that, the experiment will be extended to another type of issue: *enhancement*. Resolution times for this type of issue might show a different correlation than issues of type *defect*, or they might not show a correlation at all.

To make the implications of the correlation more clear, the results will be grouped per maintainability rating, so determine what resolution times can be expected from a system with a certain maintainability rating. This will be done for defects, and for enhancements if their resolution times are correlated

with maintainability.

Although resolution time is a useful indicator, it looks at issue handling efficiency from a certain perspective. It is useful to define more indicators that have different perspectives. For each of these indicators it needs to be determined if they are correlated with the SIG Quality Model ratings. If so, the results can be grouped per maintainability rating. Like with resolution time, this will make it easier to make assumptions about a system's issue handling based on its maintainability rating, and vice versa.

1.4 Definitions

Defect A problem in the system. Zeller [38] prefers this term over the more common “bug”. The same term is used to describe the type of issue that is reported to fix such a problem.

Enhancement An enhancement can be the addition of a new feature, or an improvement of an existing feature. Again, apart from the actual modification this term is also used to describe the type of issue that requests its implementation.

ITS *“An Issue Tracking System (ITS) is a software system that is used to manage the submission and solving of issues that arise during the life cycle of a software project.”* [18]

Issue A defect (or bug) in the software, a request for enhancement, or a task that needs to be performed on request.

Issue backlog The number of issues in an “open” state at a given point in time.

LOC Lines Of Code, a metric used to compare the sizes of different software systems.

“A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.” [8].

Software maintainability *“Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.”* [16]

Version Control System (VCS) A tool, commonly used in software development, that stores a number of files and keeps track of their change history. Every modification to a file, called a *commit*, stores the author and the time of modification.

1.5 Outline

Chapter 2 will give background information on the context of this thesis. This introduces terms that were introduced in this chapter, such as ITSs and issue life cycles. It will also give some information on the Software Improvement Group (the host organization where this thesis was written). Finally, this chapter presents the results of a literature study that was performed.

Chapter 3 gives more information on Luijten’s experiment related to issue resolution time. Chapter 4¹ describes the replication and extension of this experiment. It presents the criteria for including or excluding issues, the calculation of resolution time ratings, and the correlation results between these ratings and the SIG Quality Model ratings. After that, it repeats the experiment for defects and enhancements, and then quantifies resolution times.

Because the experiments related to issue resolution time are an extension of Luijten’s, they have to use the same dataset. This does not apply to new indicators. Chapter 5 defines a number of criteria for systems in the dataset, and then creates a new, larger dataset.

Chapter 6 and 7 define new indicators of issue handling efficiency: *enhancement ratio*, *project productivity*, and *developer productivity*. The experiments in these chapters use the extended dataset. For all indicators a correlation test is performed, to determine if they are correlated with the SIG Quality Model ratings.²

Finally, Chapter 8 answers the research questions, draws conclusions, and contains suggestions for future work.

¹This chapter has been submitted to the Software Quality Journal as a paper titled “*Faster issue resolution with higher technical quality of software*”. Unfortunately, at the time of writing it is not yet clear if and when this paper will be published.

²Another paper, “*Better issue handling with higher code quality*”, was submitted to ICSE 2011 in August 2010. This paper is based on the indicators of issue handling efficiency that are presented in this thesis (although the paper gives the indicators different names) and their relation to the SIG Quality Model.

Chapter 2

Background information

2.1 Issue Tracking Systems

An Issue Tracking System (ITS) is a software product that is used to keep track of issues that arise during the construction and use of a software system. ITSs are commonly used in professional software development [15, 30], although they are often referred to as “issue tracker” or “bug tracker”. Users of an ITS include developers, testers, and people who use the software system [10]. Reporting issues allows them to provide feedback on the system.

The information that is stored per issue in an ITS is not standardized, and depends on the ITS that is being used. Nevertheless, there are still many similarities between the fields stored by popular ITSs (JIRA¹, Bugzilla², Issuezilla³, SourceForge⁴, Trac⁵ and Google Code⁶).

All of these ITSs store the type of issue, dates when the issue was created, worked on, resolved, status, priority, resolution, who reported the issue, and to whom it is assigned. Most of these fields are mutable and are changed throughout the issue’s life cycle. The mentioned ITSs also keep track of issues’ change history, allowing the issue life cycle to be reconstructed.

¹<http://www.atlassian.com/software/jira>

²<http://www.bugzilla.org>

³Issuezilla is a modified version of Bugzilla.

⁴<http://www.sourceforge.net>

⁵<http://trac.edgewall.org>

⁶<http://code.google.com>

2.2 Issue life cycle

All ITSs use the concepts of *status* and *resolution*. Both properties are used to indicate the phase in the issue's life cycle.

The status property indicates the phase in the issue's life cycle. Figure 2.1 shows the typical life cycle. When an issue is reported it has the *new* status. This gives no guarantees that the issue will ever be worked on by a developer. After being reported, a member of the development team will look at the issue. If he considers the issue to be valid, he changes the status to *accepted* and assign it to a developer. When that developer starts working on the issue, he changes the status to *implementation*. When the work is completed the status is changed to *resolved*. After that, there is usually some verification (testing, code review) to make sure that the issue is actually fixed, after which the status is set to *closed*.

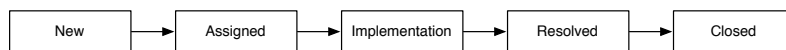


Figure 2.1: Typical issue status life cycle.

An ITS is used by many different people. This will inevitably lead to issues that cannot or should not be fixed. Some of these issues might be duplicates of issues that are already in the ITS. Others might be based on incorrect assumptions, ask for features that are outside the scope of the projects, or do not give enough information. In these cases the status is changed directly from *new* to *closed*, since no developer will ever work on it. This life cycle is shown in Figure 2.2.

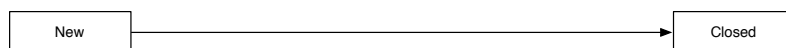


Figure 2.2: Alternative issue status life cycle.

All issue life cycles will eventually reach the *closed* status. When this status is reached, the resolution property explains the reason why the issue was closed. If the issue was resolved the resolution is *fixed*. If the issue was never implemented the resolution is set to *wontfix*, *duplicate*, or *worksforme*, depending on the reason why the issue was rejected. Different resolutions are needed because many issues are rejected, especially in projects where many people are reporting issues. In their study of the Mozilla⁷ project's issue

⁷<http://www.mozilla.com/en-US>

handling, Hooimeijer and Weimer [14] found that 36% of reported issues were rejected, and some issues have 140 duplicates.

Another study [30] found that not all developers actively update the status property. In these cases, shown in Figure 2.3, the developer picks up the issue and then directly changes the status to *resolved* once it has been implemented. In these cases it is not possible to use ITS history to determine when the developer started working on the issue.



Figure 2.3: Improper usage of the status property.

2.3 The Software Improvement Group

The Software Improvement Group⁸ (SIG) is a consultancy firm based in Amsterdam, the Netherlands. The SIG's goal is to provide insight into the technical quality of software. Organizations can use the SIG's services to assess the risks related to their software systems, or to monitor their software over a period of time.

The SIG has developed a model, the SIG Quality Model, that can be used to assess the technical quality and maintainability of a software system [13]. Using this model, a rating can be assigned to a software system's technical quality, relative to other systems. The ratings produced by the model are based on facts that are extracted from the source code through static analysis. More information about this model and how the ratings are determined is presented in Section 2.5.

The SIG Quality Model allows the SIG to provide assessments that are impartial and verifiable. The model can be applied regardless of the programming languages that were used to develop the software.

2.4 ISO/IEC 9126

The ISO/IEC 9126 international standard for Software Engineering - Product Quality [17] was released in 1991. In 2001 and 2004 the standard was further updated.

⁸<http://www.sig.eu>

The first part of the standard, ISO/IEC 9126-1, describes a model for software quality that splits quality into a number of characteristics and sub-characteristics. Figure 2.4 shows the relations between the quality characteristics and sub-characteristics. Because this study focuses on maintainability, it is highlighted in the figure.

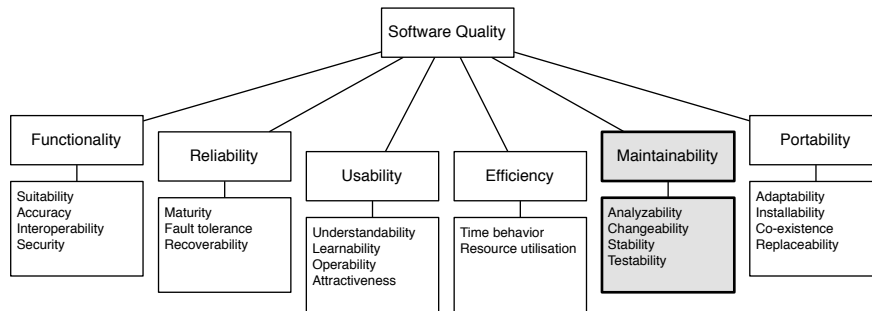


Figure 2.4: The ISO/IEC 9126-1 software quality model.

According to the standard, the sub-characteristics can be split into a number of attributes that can be measured. However, these attributes and how to measure them are not a part of the standard.

2.5 The SIG Quality Model

The SIG has created a model that maps source code properties to the sub-characteristics of maintainability [13]. The source code properties can be measured, which makes the evaluation objective and reproducible. Based on these source code properties, ratings between 0.5 and 5.5 are assigned to maintainability and its sub-characteristics. The ratings are usually presented as a rounded number between 1 and 5 stars.

The following source code properties are used:

Volume Overall size of the system.

Duplication Relative amount of code that has a textually identical clone somewhere else in the system.

Unit complexity When analyzing a unit⁹, higher complexity means more different potential scenarios of execution to take into account.

⁹The definition of “unit” depends on the programming language that is being used. For example, in Java a method is considered a unit, while in COBOL sections and/or paragraphs are used as units.

Unit size When analyzing a unit, a larger size means more source code to take into account.

Unit interfacing Size of unit interfaces. When analyzing a unit, a larger interface means more parameters to keep track of.

Module coupling Distribution of incoming calls. When analyzing a unit, more inward calls means the unit is called from a lot of places. More testing has to be done to make sure there are no unwanted side-effects.

These definitions were taken from internal documentation used within the SIG. Examples of how these properties are measured are LOC for volume, and the McCabe Cyclomatic Complexity [28] for unit complexity.

The source code properties map to the ISO/IEC 9126 maintainability sub-characteristics, which is shown in Figure 2.5. The ratings for the sub-characteristics are then used to assign a rating to maintainability itself.

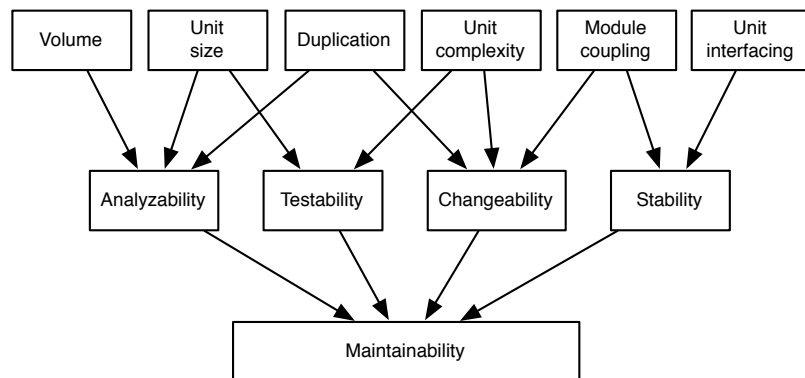


Figure 2.5: The SIG Quality Model maps source code properties to ISO/IEC 9126 sub-characteristics.

Because of the importance of maintainability (as described in Chapter 1), the ratings in the SIG Quality Model can also be seen as an indicator of the software’s technical quality in general.

2.6 Related work

This section describes the results of a literature study that was performed during the planning of this thesis. The next sections all reference existing work on topics that are related to the focus of this research.

2.6.1 The influence of software maintainability on issue handling

In 2009 Bart Luijten of the Delft University of Technology researched the relations between the SIG Quality Model and information about issue handling that can be obtained from ITS data [26, 25]. In order to conduct his experiments Luijten created a tool that extracts ITS data from a number of popular ITSs, saves it to a unified format, and attempts to link issues to source code changes.

Using this data, Luijten found a positive correlation between the SIG’s maintainability ratings and the total time an issue was in an open state. He also created a number of views that can be used to visualize ITS usage.

Our research can be seen as an extension of Luijten’s work. A more detailed description of Luijten’s work is presented in Chapter 3.

2.6.2 ITS usage

Nicolas Bettenburg of the Saarland University has researched the usage of ITSs for three large open source systems [4]. After analyzing bug reports and asking both developers and users to complete a survey, he concluded that “there is a mismatch between what [information] developers consider most helpful and what users provide.”

The survey asked both developers and users to rank issue properties in order of relevance. It also asked users how easy or difficult they found it to provide this information.

Developers considered the steps to reproduce the problem and stack traces (error messages) the most helpful, but users generally found it very difficult to provide this information. Many issue properties that were often provided by users were considered not helpful by developers.

Even though users were not always able to report this information, they did have a similar view of what developers would consider helpful. The largest difference is including a screenshot in the issue, which was considered helpful by developers but not by users. Bettenburg does not provide an explanation for this difference in opinion.

Developers and users were also asked about their experiences in using ITSs. This revealed that the real priority of an issue is determined by many factors, and not necessarily by the *priority* property. If the issue’s description is vague, if the person who reported the issue is rude, or if the issue is very

difficult to reproduce there is less chance that the issue will be resolved.

Public debate was considered the most common misuse of an ITS. Issues are often used for discussing the importance of various features. Comments are added with different reasons of why a particular enhancement is needed, making it more difficult to determine what the issue is actually about.

Based on these results, Bettenburg developed a metric to assign a quality rating to an issue. Only the quality of the information provided in the issue is rated, the severity of the actual problem is not taken into account.

Unfortunately this rating system is not directly relevant to issue handling efficiency, since it only assesses the quality of issues being *reported*, and not how issues are *handled*. Even though it is not directly related, the survey still provides valuable information about ITS usage.

2.6.3 Extracting ITS data

A tool very similar to the one developed by Luijten was developed by Annie Tsui Tsui Ying during her research towards change patterns [37]. Ying's tool also extracts both ITS and VCS data, and attempts to create links between them. However, the model in which this tool stores ITS data is less suited for analyzing issue handling efficiency, and entirely focused at linking source code, ITS data and version control system data. It also only supports the Bugzilla ITS, and stores data in a model that directly maps to Bugzilla, making it difficult to extend the tool with support for other ITSs that use different issue properties.

BugCrawler [9] is a tool that visualizes the evolution of a system. One of the sources for this information is a system's Bugzilla history. Because this tool is not yet publicly available it cannot be determined what model is used to store ITS data, and if it can be extended to support other ITSs.

2.6.4 Linking ITS data to source code changes

One problem with looking at ITS data is that it can be difficult to make conclusions on how an issue actually impacts the software. There have been several studies [24, 11, 2, 6, 37, 12] on how to link ITS data to the commit history of a VCS, and to the source code itself. These studies all conclude that this linkage is extremely difficult and produces many false negatives.

The most straightforward method of linking an issue to a commit is when the issue number is mentioned in the commit message. Unfortunately, the large

majority of researched projects do not follow this approach. This means that the only way to link issues and commits is by comparing the text of the commit message to the text of the issue. Ricardo Lippolis researched four projects using this approach, and concludes that linking based on words does not produce results that are good enough to trace change requests through ITS usage and commit messages [24].

Because not all issues can be traced back to the source code, there is a large chance of bias if any conclusions are made on the small part of issues that can be tracked back to the source code. According to Bird et al. [5], issue priority, experience of the developer to whom the issues is assigned, and the point in the project’s release cycle all have an influence on the traceability between issue and source code.

Because of these problems in linking issues to source code, this study will not attempt to create or use such links.

2.6.5 Issue life cycle visualizations

Section 2.2 uses simple figures to visualize an issue’s life cycle. These figures show the transitions between states, but do not give any information on how long the issue remains in one of these states.

D’Ambros et al. [10] propose a visualization of an issue’s life cycle that they call a “bug watch”. An example of this visualization is shown in Figure 2.6. The colors of the segments represent the issue’s state, the size of the segments give an indication how long the issue remained in that state.

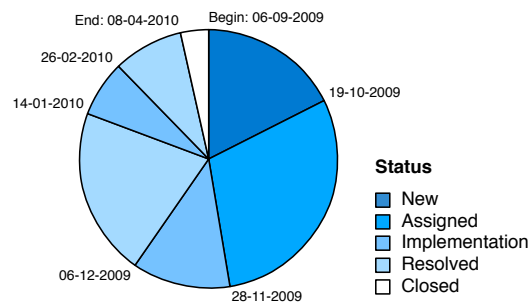


Figure 2.6: Example of a “bug watch” visualization..

Chapter 3

Preceding experiment

This study is largely based on a previous experiment that was performed by Bart Luijten of the Delft University of Technology in 2009 [26, 25]. This chapter gives an overview of this experiment, and the concepts that were introduced by Luijten.

3.1 Unified model for ITS data

The main problem with comparing data from different ITSs is that all of them use different issue properties. Luijten identified a number of issue properties that were present (although sometimes with different names) in all ITSs that were used by projects in his dataset. He then created a model that stores these properties in a consistent way. The model also keeps track of issues' change history.

The model makes a distinction between immutable issue properties that are set when the issue is created, and mutable issue properties whose value can be changed throughout an issue's life cycle. A set of all mutable issue properties at a specific point in time is an issue's *state*. A timestamp is attached to each state, making it easy to determine the value of all the issue's properties at any given time.

The model identifies the following immutable issue properties:

ID Unique identifier for the issue.

Submitter The person who originally reported the issue.

Submit date The date and time at which the issue was originally reported.

The model also identifies the following mutable issue properties:

Type Describes the type of issue. One of *defect*, *enhancement*, *task* or *patch*.

Description A textual description of the issue. If the issue is a defect, the description typically includes the steps to reproduce it. For enhancements, the description may include the rationale for why it is needed.

Priority Priority of solving the issue. Depending on the ITS these are either numeric or textual. When textual some typical values are *critical*, *major*, *minor*, and *trivial*.

Status Describes the current point in the issue's life cycle. Examples of values are *new* and *closed*.

Resolution Similar to status, but is set only for closed issues and gives an explanation of why the issue was closed. Examples of values are *fixed* and *duplicate*.

Assignee The person whom the issue is assigned to.

Component The subsystem, module, or component that is affected by the issue. Note that an issue can affect multiple components, making this field somewhat unreliable.

3.2 ITS Data extraction tool

Luijten created a tool that is capable of extracting ITS data from a number of different ITSs and storing the extracted data in the format described in Section 3.1. The tool originally supported four ITSs¹, but allows for more to be added.

The tool also extracts the commit history of the VCS (currently only Subversion² is supported). This is done so that commits can be linked to issues, making the effect a particular issue has on the source code clearer.

Finally, the tool extracts the source code itself, calculates the SIG Quality Model ratings based on the extracted source code, and then stores these ratings in a database.

¹JIRA, BugZilla, IssueZilla, and SourceForge. URLs for these products can be found in Section 2.1.

²<http://subversion.apache.org>

3.3 Issue backlog

The issue backlog for a project consists of all the issues that are in an “open” state³ at a given point in time. Having a large backlog makes it more difficult to determine what issues are important and what issues can be fixed easily. It can also be an indicator of poor maintainability. With poorly maintainable software resolving defects takes longer [7], leaving less time for working on enhancements.

3.4 Issue resolution time

Issue resolution time is the total time an issue has been in an “open” state. Usually this is the time from the moment the issue was opened to the moment it was resolved. If the issue is reopened after it has been closed the resolution time is the total of the periods that it was open.

This metric can be used to determine the efficiency of issue handling for a project. However, it does not give an explanation whether issue handling is efficient because of high technical quality or because of a good development process. Similarly, it is also impossible to point out a single cause for poor resolution times. Long issue resolution times can be caused by under-capacity of the development team, or simply because many users are reporting low-priority issues and the large number of issues causes the development team to ignore some of these issues for longer periods of time. In his study of open source software development Audris Mockus [30] concluded that for the Apache HTTP Server project “few developers keep an active eye on the bug database”.

Although acknowledging these flaws in the metric, it can still be assumed that highly maintainable software will usually make it easier to resolve issues, thus increasing issue handling efficiency and reducing resolution times. In order to assess this, resolution times for projects with a known maintainability level have to be compared.

3.5 Issue resolution time rating

Resolution times are calculated per issue, but it can be useful to compare them on a project level. This could be done by taking the average or median

³ An issue is in an open state if the status property is set to *new*, *assigned*, or *implementation*.

resolution time for a project, but because the distribution of resolution times is not normal this is not a fair comparison.

To solve this problem, Luijten created “risk profiles” based on all resolution times. Using these risk profiles a rating between 0.5 and 5.5 can be assigned to a project. These ratings can then be used to compare projects based on issue solving efficiency.

Luijten also tested the correlation (using Spearman’s rank correlation [34]) between the resolution time risk profile ratings and the SIG Quality Model ratings, of which the results are shown in Table 3.1. He found a positive correlation for all ratings, apart from Unit Interfacing for which the test results were not significant enough.

Defect resolution time vs.	ρ_s	p-value
Volume	0.29	0.003
Duplication	0.31	0.002
Unit size	0.51	0.000
Unit complexity	0.51	0.000
Unit interfacing	-0.14	0.897
Module coupling	0.51	0.000
Analyzability	0.51	0.000
Changeability	0.64	0.000
Stability	0.41	0.000
Testability	0.53	0.000
Maintainability	0.62	0.000

Table 3.1: Luijten’s correlation results for defect resolution time.

Chapter 4

Issue resolution time

This chapter describes the experiments we performed in relation to issue resolution time. We start by replicating Luijten’s experiment that was described in Chapter 3. After that, we extend the experiment and quantify the results.

4.1 Criteria for issue resolution time

Issue resolution time is the total time an issue is an open state. The resolution time is not simply the time between the issue being reported and the issue being resolved, since the issue can be set to open again after it has first been closed. Figure 4.1 shows a possible life cycle of an issue, and highlights the period which is counted as resolution time.

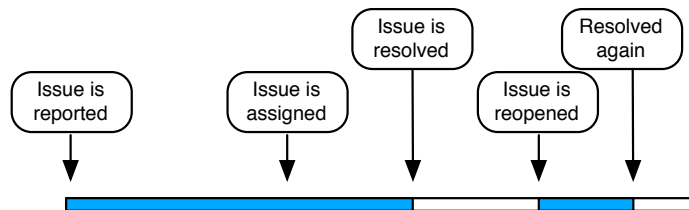


Figure 4.1: The life cycle of an issue. Resolution time is the period marked in orange.

From the perspective of measuring effort, it would seem better to start counting once the status is set to *assigned*. If an issue is never assigned, it will not consume any resources. Unfortunately the projects in the dataset do not make consistent use of the status property. They suffer from the

same problem that was identified by Audris Mockus when observing ITS usage in the Apache HTTP server project [30], and that was mentioned in Section 2.2: issues are frequently resolved without having been in the *assigned* or *implementation* status. Because these status values are not used it is impossible to accurately determine when an issue was picked up by a developer. This is the reason why we start measuring resolution time when the issue is reported.

4.2 The dataset

The replication of Luijten’s experiment will be performed on the same dataset that was used in the original experiment, which is shown in Table 4.1.

Project	Snapshots	Main language	LOC	ITS	Issues
Abiword	1	C++	332K	Bugzilla	10,941
Ant	20	Java	122K	Bugzilla	5,192
ArgoUML	20	Java	171K	Issuezilla	5,789
Checkstyle	22	Java	45K	SourceForge	612
Hibernate-core	1	Java	145K	JIRA	4,009
JEdit	1	Java	54K	SourceForge	3,401
Spring Framework	21	Java	119K	JIRA	5,966
Subversion	1	C	218K	Issuezilla	3,103
Tomcat	19	Java	164K	Bugzilla	644
Webkit	1	C++	1.3M	Bugzilla	22,016
	107				61,673

Table 4.1: The dataset used in the experiments.

As explained in Section 2.2, the fact that an issue is reported does not necessarily mean that it will ever get resolved. Such issues do not have an influence on issue handling efficiency, and should not be taken into account while determining the issue resolution times for a snapshot. Therefore, these issues have to be excluded from the dataset before the experiment can be performed.

We will apply Luijten’s original criteria for removing issues from the dataset. The most obvious criterion for removing issues is a resolution of *wontfix*, *duplicate*, or something similar (these resolutions were explained in Section 2.2), as these issues were never worked on by a developer. The other criterion used by Luijten is minimum snapshot size. To establish a reliable resolution time rating for a snapshot, it needs to be represented by at least five issues.

One of Mockus’ observations was that some developers do not keep a close

eye on the ITS. This can lead to discrepancies in the data, issues have been resolved but are still marked as open in the ITS. To determine if this problem exists in our dataset we visualize the number of resolved issues per month for each system. If there are large fluctuations in these numbers, it might be an indication of unusual ITS usage and the underlying ITS data needs to be examined.

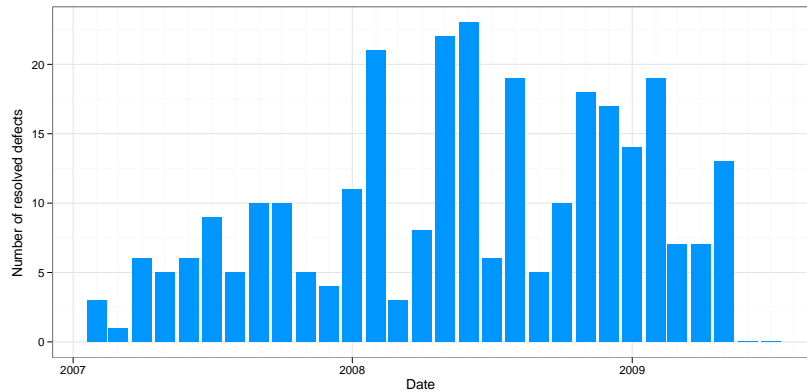


Figure 4.2: Number of resolved defects per month for Tomcat.

Figure 4.2 shows the number of resolved defects per month for the Tomcat¹ project. As can be seen, this number is very stable (between 5 and 20) for a period of over two years.

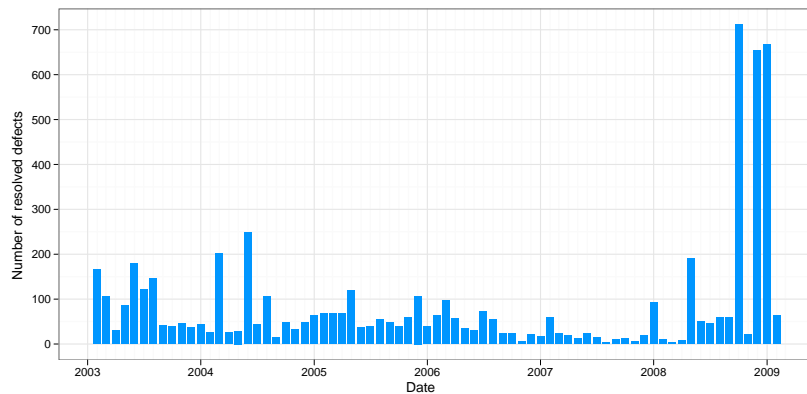


Figure 4.3: Number of resolved defects per month for ArgoUML (before cleaning).

Figure 4.3 shows the number of resolved defects, but for the ArgoUML² project. The number of resolved defects is usually below 100, except for late

¹<http://tomcat.apache.org>

²<http://argouml.tigris.org>

2008 where there were suddenly more than 600 defects resolved per month for a period of three months. Such an increase seems unusual, and might indicate a problem with ITS usage.

Upon manual inspection of the issues that were resolved in late 2008, all defects that were bulk-closed were closed with the same comment. Some examples:

“The solution to this issue is included in the stable release 0.26.2. (...) If you, when you test this, find that the issue is not solved, create a new issue and mention this issue in the description or as a reference.”
(30 December 2008, 796 times)

“This issue was resolved a long time ago.”
(21 September 2008, 454 times)

These comments clearly indicate unusual ITS usage. For these issues it is not possible to determine the date on which they were resolved, as the bulk-close might have happened months after the issue was solved. For this reason, these issues will be removed from the dataset.

4.3 Resolution time ratings

It is interesting to compare snapshots in terms of issue resolution times. Comparing snapshots by taking the average or median resolution time would be inappropriate, since the distribution of resolution times is not normal (as can be seen in Figure 4.4).

Taking the same approach as Luijten, we create a rating system for issue resolution times for a particular project. This rating system splits resolution times into four risk profiles [27], which are based on the 70th, 80th, and 90th percentiles. Using the risk profiles on the projects in the dataset gives the results that are shown in Table 4.2.

Note that the displayed values are not the actual 70th, 80th and 90th percentile. The values were rounded to the closest human-readable period to make their meaning clearer.

These risk profiles can be used to assign a rating between 0.5 and 5.5 to the issue resolution times for a particular project. Ratings are usually rounded

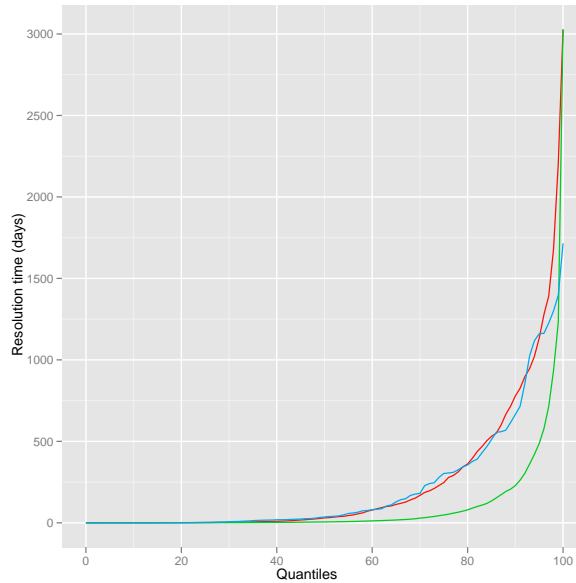


Figure 4.4: Percentile plots of defects (green), enhancements (red) and patches (blue).

Category	Thresholds	
Low	0 - 28 days	(4 weeks)
Moderate	28 - 70 days	(10 weeks)
High	70 - 182 days	(6 months)
Very high	182 days or more	

Table 4.2: Risk profile thresholds for defect resolution time.

and displayed as a number of stars between 1 and 5. The requirements to obtain each of these star ratings are shown in Table 4.3.

The number of snapshots per resolution time rating for both defects and enhancements is shown in figures 4.5 and 4.6. These distributions are comparable, meaning that the correlation test that Luijten performed for issues of type *defect* can also be performed for issues of type *enhancement*.

4.4 Correlation results for defects

The experiment can now be reproduced with cleaned dataset. Recalculating Spearman's rank correlation coefficients [34] with this dataset and the 2010 version of the SIG Quality Model produces the results shown in Table 4.4.

The values on the left are from the original experiment, the ones on the right

Rating	Moderate	High	Very high
5 stars	8.3%	1.0%	0.0%
4 stars	14%	11%	2.2%
3 stars	35%	19%	12%
2 stars	77%	23%	34%

Table 4.3: Resolution time ratings with risk profile criteria.

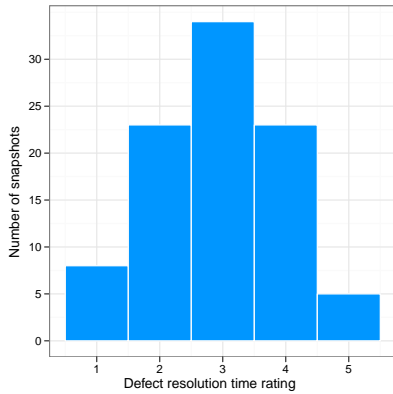


Figure 4.5: Histogram for defects.

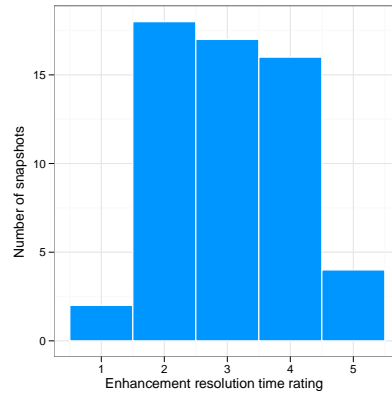


Figure 4.6: Histogram for enhancements.

(in bold) are from the experiment with the extended and cleaned dataset.

All of the measured correlations are slightly stronger in the results of the new experiment. Further investigation shows that this improvement is mainly caused by the removal of issues closed in bulk in the ArgoUML project. The results for Unit Interfacing are still not at a 99% significance level, even after cleaning the data, although it has improved a lot in comparison to Luijten's results.

4.5 Correlation results for enhancements

Section 4.4 showed a positive correlation for issues of type *defect*, but the experiment can be adapted to issues of type *enhancement*. These might show a stronger correlation, since enhancements usually have a larger scope than defects. The experiment uses the same dataset as the experiment for defects. The results are shown in Table 4.5.

The correlations with maintainability and its sub-characteristics are slightly lower compared to the correlations for defects. The results for the source code property show a much higher correlation with Volume. This indicates

Defect resolution time vs.	ρ_s	p-value	ρ_s	p-value
Volume	0.29	0.003	0.37	0.001
Duplication	0.31	0.002	0.35	0.001
Unit size	0.51	0.000	0.56	0.000
Unit complexity	0.51	0.000	0.56	0.000
Unit interfacing	-0.14	0.897	0.27	0.046
Module coupling	0.51	0.000	0.56	0.000
Analyzability	0.51	0.000	0.62	0.000
Changeability	0.64	0.000	0.70	0.000
Stability	0.41	0.000	0.49	0.000
Testability	0.53	0.000	0.58	0.000
Maintainability	0.62	0.000	0.65	0.000

Table 4.4: Defect resolution time correlation results (original and **updated**).

Enhancement resolution time vs.	ρ_s	p-value
Volume	0.71	0.000
Duplication	0.11	0.190
Unit size	0.50	0.000
Unit complexity	0.57	0.000
Unit interfacing	0.15	0.126
Module coupling	0.65	0.000
Analyzability	0.53	0.000
Changeability	0.51	0.000
Stability	0.48	0.000
Testability	0.55	0.000
Maintainability	0.56	0.000

Table 4.5: Enhancement resolution time correlation results.

that it is much easier to add functionality to a small software system than to a large one. Like with defects, the results for Unit Interfacing are not significant.

4.6 Quantification of issue resolution times

Now that we know that a positive correlation between issue resolution time and the maintainability ratings (still) exists, we can start determining how much shorter resolution times for highly maintainable software systems are, compared to systems of a lower maintainability.

The snapshots are grouped per maintainability rating. For each of these groups, the median defect resolution time and median enhancement resolution time is calculated. The median is used instead of the mean because the distribution of resolution times (shown in Figure 4.4) is not normal. The re-

sults are shown in Table 4.6 and depicted in Figure 4.7. There are no results for very low and very high maintainability ratings because no snapshots in the dataset have such a rating.

Maintainability rating	Snapshots	Median defect resolution time (days)		Median enhancement resolution time (days)	
4.5	1	4.1	(÷ 2.4)	4.3	(÷ 2.0)
4.0	5	6.5	(÷ 1.5)	6.7	(÷ 1.3)
3.5	34	8.8	(÷ 1.1)	7.9	(÷ 1.1)
3.0	29	9.8		8.7	
2.5	19	17.0	(× 1.7)	15.9	(× 1.8)
2.0	3	22.4	(× 2.3)	26.6	(× 3.1)
1.5	3	53.4	(× 5.4)	53.4	(× 6.1)

Table 4.6: Defect and enhancement resolution time per maintainability rating.

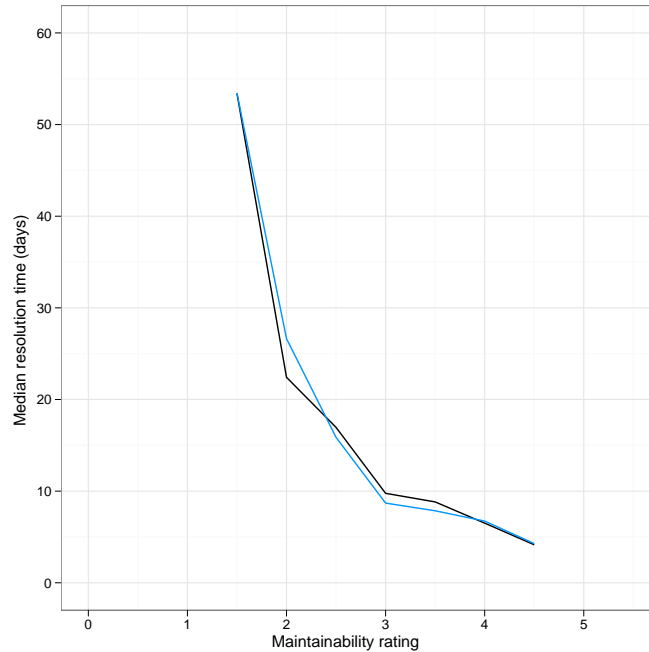


Figure 4.7: Defect (dark) and enhancement (light) resolution time per maintainability rating.

These results show that defects and enhancements behave very similar in terms of resolution time, regardless of the system’s maintainability rating. The difference between different maintainability ratings is quite large. For systems with a maintainability rating of 1.5 it takes more than five times as long to solve an issue, compared to systems with a maintainability rating of 3.0.

4.7 Discussion

Replication of Luijten’s experiment resulted in slightly stronger correlations. The difference with the original experiment’s results can be explained by the cleaning of the dataset. Issues that were handled in a unusual way had a negative influence on the original results.

Removing these issues has also improved the significance for Unit Interfacing, but it is still below the desired level of 99%. If we would accept a 95% significance level, Unit Interfacing does show a correlation with defect resolution time, although lower than the other source code properties.

The correlations between enhancement resolution time and the SIG Quality Model ratings are comparable but slightly lower than those for defects. Like with defects, the results for Unit Interfacing are not significant. Unlike defects, the results for Duplication are also not significant. One possible explanation for this lack of significance would be that the number of enhancements (around 3,000) in the dataset is much lower than the number of defects (around 11,000). However, the p-values for the other results are not lower than those for defects, making this explanation unlikely.

It can be argued that the exact resolution times are largely dependent on the type of system, and that the quantification results cannot be generalized to systems outside the dataset. The used programming language might also have an influence, since studies have shown that this has an influence on the number of defects that can be solved within a certain time period [29]. This is a valid point, but even though the actual resolution times might differ for systems with other characteristics than those in the dataset, the ratio between resolution times for different maintainability ratings is unlikely to change.

The quantification results show large differences in median resolution times between maintainability ratings. Figure 4.8 shows the median resolution time for each snapshot in the dataset. Some snapshots have median resolution times of more than two years days, which seems very long. However, examining the underlying ITS data does not show unusual ITS usage. Large numbers of issues with very long resolution times is something that also happens in other projects. A study performed in 2006 shows that the median resolution time for the PostgreSQL³ project is around 200 days [20]. According to another study by D’Ambros et al. [10] the average resolution time for issues in the Mozilla⁴ project is 523 days. Issues with very long resolution times can therefore be considered common in software development,

³<http://www.postgresql.org>

⁴<http://www.mozilla.org>

erty *duplication* were also not significant. For both types of issue, the correlation with maintainability was stronger than with its sub-characteristics. The correlation between maintainability and resolution times for defects is slightly stronger than for enhancements.

Resolution times were also quantified (RQ1.1 and RQ2.1), by taking the median resolution time for both defects and enhancements for each of the maintainability ratings. Compared to 3-star systems, 2-star systems take 2.3 times as long to fix a defect, and 3.1 times as long to implement an enhancement. 4-star systems fix defects 1.5 times faster, and implement enhancements 1.3 times faster.

Chapter 5

Extending the dataset

The experiments described in Chapter 4 was based on the preceding experiment related to issue resolution time. Because of the relation to Luijten's experiment, these experiments had to use the same dataset. Additional indicators that are going to be defined do not have this constraint. If the new experiments use a dataset with more systems and snapshots, their results will be more reliable.

5.1 Criteria for systems

The new dataset will be based on the original one. If additional systems are to be added, they should be comparable to the systems from the original dataset. Next, we establish the of criteria that systems should conform to:

- The system is developed in an open source project that uses a public ITS.
- Because the large majority of systems in the original dataset use Java as primary programming language, this should also be the case for new systems.
- The system's size (in LOC) is at least 65% of the smallest system from the original dataset.
- The system's size (in LOC) is at most 200% of the largest system from the original dataset.
- The number of issues in the system's ITS is at least 65% of the lowest number of issues from the original dataset.

5.2 The new dataset

33 new systems were considered, but only 7 met all criteria that were defined in Section 5.1. The following systems were added:

- Apache HTTPD¹
- Compiere²
- HSQLDB³
- iBatis⁴
- Limewire⁵
- Log4j⁶
- Xerces⁷

Between 1 and 8 snapshots were added for each system. The goal was to have an equal time period (roughly a year) between snapshots. The Mantis ITS used by Compiere was not supported by Luijten's tool, so the tool was extended to support importing data from this ITS.

Also, more snapshots of the same systems were added to the dataset (using a more equal distribution of snapshots between systems was one of Luijten's recommendations for future work).

Looking at the dataset, Webkit is four times bigger than the second largest system (Abiword) in terms of LOC. This makes it hard to compare Webkit to the other systems, and it is therefore removed from the dataset.

¹<http://httpd.apache.org>

²<http://www.compiere.com>

³<http://hsqldb.org>

⁴<http://www.mybatis.org> (the project has been renamed to MyBatis in May 2010)

⁵<http://www.limewire.com>

⁶<http://logging.apache.org/log4j>

⁷<http://xerces.apache.org/xerces-j>

These changes lead to the dataset presented in Table 5.1. This dataset will be used for the experiments with new indicators.

Project	Snapshots		Main language	LOC	Issues	Maint. rating
	Original	Extended				
Abiword	1	3	C++	332K	10,941	1.9
Ant	20	20	Java	122K	5,192	3.4
ArgoUML	20	20	Java	171K	5,789	3.2
Checkstyle	22	22	Java	45K	612	3.8
Hibernate-core	1	5	Java	145K	4,009	2.9
JEdit	1	4	Java	54K	3,401	2.7
Spring Framework	21	21	Java	119K	5,966	3.9
Subversion	1	5	C	218K	3,103	1.9
Tomcat	19	19	Java	164K	644	2.7
Webkit	1	N/A	C++	1.3M	22,016	1.9
HSQLDB	N/A	5	Java	65K	998	2.4
iBatis	N/A	6	Java	34K	777	3.0
Log4j	N/A	6	Java	31K	1,201	3.8
Apache HTTPD	N/A	4	C	138K	4,237	2.1
Compiere	N/A	1	Java	422K	499	2.0
Limewire	N/A	2	Java	343K	3,817	3.8
Xerces	N/A	8	Java	168K	426	2.2
16	107	151			61,673	

Table 5.1: The dataset used in the experiments.

The LOC and maintainability rating displayed in this table were taken from the most recent snapshot for each system. In reality, each snapshots has its own LOC and maintainability.

Chapter 6

Enhancement ratio

In most open source projects it is required to create an issue in the ITS for every modification that is made to the source code. Therefore, the categories of development activity are the same as the categories of issues: *defect*, *enhancement*, *task*, and *patch*. Since defects are a form of *corrective maintenance* [7], it is preferable that there are as few defects as possible, which means that developers will spend more time working on enhancements (a form of *perfective maintenance*).

To determine if the maintainability of the software has an influence on the ratio between working on defects and working on enhancements, we form the following hypothesis:

[H1] Developers of software systems with higher maintainability will implement more enhancements (relative to their total effort), compared to developers of systems of lower maintainability.

This hypothesis intentionally ignores issues of types *task* and *patch*. Tasks are usually one-time activities that need to be performed on request, and do not have an impact on the system's source code. A patch is only related to the source code, it is not possible to determine if the patch is used for *corrective maintenance* or *perfective maintenance*.

In order to verify the hypothesis we define *enhancement ratio* as the following:

$$\text{enhancement ratio} = \frac{RE}{RD + RE} \times 100\%$$

where RE is the number of resolved enhancements for a certain time period, and RD is the number of resolved defects for that same time period.

For this experiment, only snapshots with at least five resolved defects and at least five resolved enhancements are taken into account. Snapshots with less than five issues could produce very high or very low enhancement ratios, which would not be realistic since no conclusions should be made based on so few issues.

6.1 Results

Figure 6.1 shows the percentage of enhancements (dark) and defects (light) out of all resolved issues for each snapshot in the dataset. The ratio between working on defects and working on enhancements is quite stable between snapshots of the same system. The Spring Framework is consistently scoring the highest percentage of enhancements, and Subversion is consistently scoring low. Figures 6.2 and 6.3 show the enhancement ratios for these two systems per month. These figures also indicate that the enhancement ratio for a system stays fairly stable.

As with issue resolution time, a Spearman test is performed to test the correlation between the enhancement ratio and the SIG Quality Model ratings. Table 6.1 shows the results of this test.

Enhancement ratio vs.	ρ_s	p-value
Volume	0.05	0.645
Duplication	0.09	0.239
Unit size	0.52	0.000
Unit complexity	0.47	0.000
Unit interfacing	0.22	0.031
Module coupling	0.40	0.000
Analyzability	0.33	0.002
Changeability	0.33	0.002
Stability	0.37	0.000
Testability	0.50	0.000
Maintainability	0.39	0.000

Table 6.1: Enhancement ratio correlation results.

These results show a positive correlation between enhancement ratio and maintainability and all of its sub-characteristics. The source code properties *unit size*, *unit complexity*, *module coupling*, and *unit interfacing* also show a positive correlation. The results for *volume* and *duplication* are not significant.

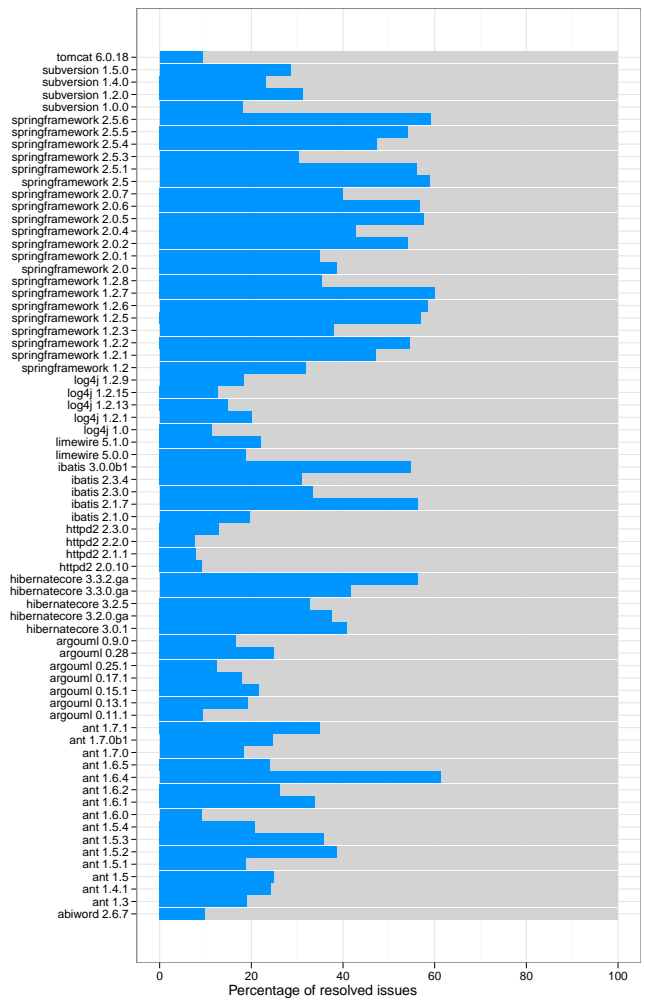


Figure 6.1: Percentage of enhancements (dark) and defects (light).

6.1.1 Quantification

Since enhancement ratio is correlated with maintainability, the results can be quantified to determine the enhancement ratio associated with a particular level of maintainability. This is done by calculating the median enhancement ratio for each of the maintainability star ratings. We do not group per half star, like we did with resolution times in Chapter 4, because resolution times are calculated per issue, while enhancement ratios are calculated per snapshot. Therefore, grouping per half star would give results based on very little data.

Enhancement ratio is an *interval scale*. Ratios between numbers on such

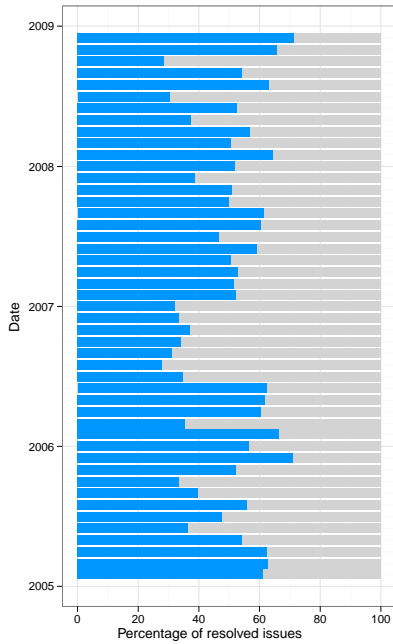


Figure 6.2: Spring Framework

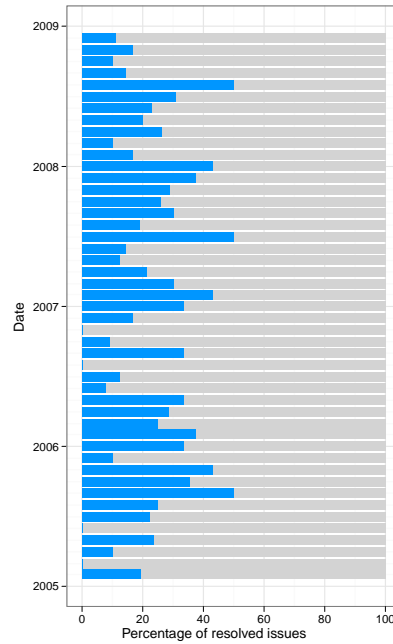


Figure 6.3: Subversion

a scale are not meaningful, so we cannot quantify enhancement ratio by calculating the multiplication factors between values, as we did in Chapter 4. Table 6.2 therefore shows the differences between enhancement ratios for the different maintainability ratings.

Maintainability rating	Snapshots	Median enhancement ratio
5 stars	0	N/A
4 stars	31	38.7% + 13.1%
3 stars	30	25.6%
2 stars	9	12.9% - 12.7%
1 star	0	N/A

Table 6.2: Median enhancement ratio for each maintainability rating.

Figure 6.4 depicts enhancement ratios for different maintainability ratings. This figure shows a larger difference in enhancement ratio between 2 and 3 stars than between 3 and 4 stars. These differences are quite similar to those of resolution time (shown in Section 4.6). It is also interesting to point out that even with a maintainability rating of 4 stars the enhancement ratio is less than 50%, meaning that the majority of work is spent on fixing defects. No results are available for maintainability ratings of 1 and 5 stars, as the dataset does not contain any snapshots with these ratings.

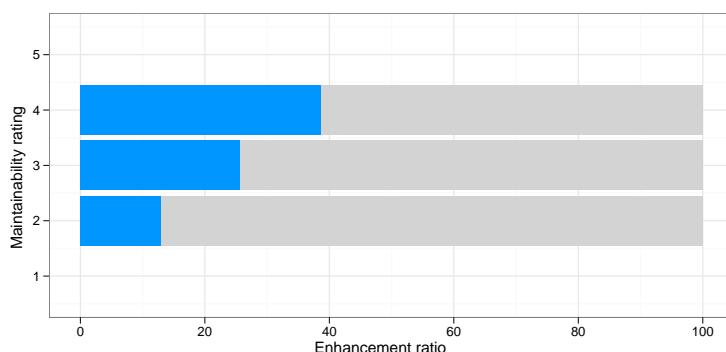


Figure 6.4: Median enhancement ratio for each maintainability rating.

6.2 Discussion

The *unit interfacing* source code property has a much lower correlation than the others, although this result has a reduced level of confidence. Nevertheless, we can still assume that this result is reliable, and that this property has a very limited influence on a system’s enhancement ratio.

Enhancement ratio ignores the time developers spend working on issues of type *task* and *patch*. We do not look at tasks because they usually do not affect the source code. Patches are ignored because it cannot be determined if a patch is used to fix a defect or enhance the software. Patches are only used by two systems in the dataset, and even there they account for less than 10% of issues. We therefore do not expect the exclusion of patches to have had an influence on the correlation results.

One possibly erroneous assumption that all source code changes are linked to issues in the ITS. Studies [30, 24] have shown that this is not always the case. We acknowledge this risk, but we assume that in the majority of cases source code changes are linked to issues, as nearly all systems in the dataset request this from their developers.

It could also be argued that the phase in the system’s life cycle influences the need for working on enhancements. Some systems will eventually reach a phase where almost no enhancements are requested, because all desired functionality is already present. One system in the dataset which is both very mature and has a low enhancement ratio is Subversion¹. Examining the underlying ITS data shows that there are more than 200 open enhancements with medium or high priority. This makes it hard to argue that the system meets all functional requirements.

¹<http://subversion.apache.org>

Another possible confounding factor is the type of system. Some of the systems in the dataset are libraries, and have a limited and well defined scope. Others are applications with a large feature set, which makes it more difficult to determine if a proposed enhancement is within the scope of the system. Future research should determine if different types of systems have different characteristics in terms of the number of defects and enhancements that are resolved. If this is true, the experiment has to be repeated for a new dataset which only contains systems of the same type.

Finally, the enhancement ratio does not take the ratio between *reported* defects and enhancements into account. This is done for reliability. If an enhancement is reported, there is no guarantee that it will ever get implemented. The enhancement might only be useful for the person who reported it, it might be outside of the scope of the project, or it might have very low priority. Looking at the resolved enhancements does not suffer from these problems, because it is certain that those issues were valid.

6.3 Conclusion

The results show a positive correlation between the SIG Quality Model's maintainability rating and enhancement ratio. A similar correlation exists between maintainability's sub-characteristics and enhancement ratio, and for four of the six source code properties. For the other two source code properties the results are not significant. Although this lack of significance prevents a conclusion about the relation between the source code properties and the enhancement ratio, it can be concluded that there is a relation between maintainability and enhancement ratio. [H1] can therefore be confirmed.

Chapter 7

Project and developer productivity

When it comes to ITS usage, productivity can be seen as the number of issues that were resolved in a certain time period. There are several factors that could potentially make it difficult to compare these numbers between one system and another:

- The number of developers working on a system might be higher, leading to more resolved issues but not necessarily higher productivity per developer.
- Larger systems have more functionality, and thus more features against which issues can be reported.

To measure productivity between systems with different numbers of developers we make a distinction between *project productivity* (the combined productivity of all project members) and *developer productivity* (the productivity per project member).

The productivity of systems of different sizes can be compared by dividing the number of resolved issues by LOC. This defines project productivity as the following:

$$\text{project productivity} = \frac{\text{resolved issues per month}}{KLOC}$$

We use KLOC (1 *KLOC* = 1000 *LOC*) instead of LOC to get nicer numbers. Developer productivity can be calculated from project productivity:

$$\text{developer productivity} = \frac{\text{project productivity}}{\text{developers}}$$

In theory, lack of maintainability can be compensated with more manpower to keep (project) productivity high. Based on this theory, we form the following two hypotheses:

- [H1] There is no correlation between the maintainability of a software system and the *project productivity* for that system.
- [H2] There is a correlation between the maintainability of a software system and the *developer productivity* for that system.

Project productivity and developer productivity cannot be measured properly if there are too few resolved issues for a snapshot. For this reason all snapshots with five or less resolved issues are removed from the dataset.

7.1 Determining the number of developers

In order to calculate developer productivity, the number of developers working in a certain time period needs to be known. There are two methods to obtain this number. The first is from the *assignee* issue property. The second is by looking at the VCS's commit history, and counting the developers that made a commit.

None of these methods will produce an accurate number. In some projects, all reported issues are assigned to a lead developer, who then reassigns the issue when it is accepted. If the issue is resolved without updating its state (as described in Section 2.2) this reassign might never happen, which would lead to an assumed number of developers that is too low.

Counting the number of committers in the VCS history can also produce a number that is too low. In some projects not all developers have commit rights. Developers without commit rights send their modifications to a developer that does have them, who then reviews the changes and commits them.

So, both methods could produce a number too low. To determine what method is most accurate, we use both methods and see which one consistently produces the higher number. The results are shown in Figure 7.1, and show that using the VCS history has more chance of including all developers. We will therefore use this method to count the number of developers working on a system.

	Count	Percentage
Using <i>assignee</i> property produces a higher number	44	29%
Using VCS history produces a higher number	95	63%
Numbers are equal	12	8%

Table 7.1: Two methods of determining the number of developers.

7.2 Results

Figure 7.1 and Figure 7.2 show the project and developer productivity per month for all snapshots in the dataset. The values for developer productivity show larger differences between snapshots, with very high values for Hibernate 3.2.0 and Spring Framework 1.2.2. Inspection of the underlying ITS data does not show unusual ITS usage for these snapshots. In fact, it confirms the productivity of the developers. After the 3.2.0 release, the Hibernate developers resolved more than 600 issues in the next nine months, most of them on the same day as they were reported.

	Project productivity		Developer productivity	
	ρ_s	p-value	ρ_s	p-value
Volume	0.46	0.000	0.60	0.000
Duplication	0.34	0.000	0.44	0.000
Unit size	0.46	0.000	0.59	0.000
Unit complexity	0.53	0.000	0.62	0.000
Unit interfacing	0.27	0.002	0.38	0.000
Module coupling	0.46	0.000	0.53	0.000
Analyzability	0.50	0.000	0.66	0.000
Changeability	0.57	0.000	0.66	0.000
Stability	0.39	0.000	0.51	0.000
Testability	0.48	0.000	0.61	0.000
Maintainability	0.57	0.000	0.68	0.000

Table 7.2: Project productivity and developer productivity correlation results.

Like with the other indicators, it needs to be determined if project productivity and developer productivity are correlated with the SIG Quality Model ratings. We therefore perform another Spearman test, of which the results are presented in Table 7.2. Both project productivity and developer productivity show a positive correlation with maintainability, all of its sub-characteristics, and all source code properties. The correlations for developer productivity show a similar trend to those for project productivity, but are consistently higher. For both indicators Duplication and Unit Interfacing show a weaker correlation than the other source code properties.

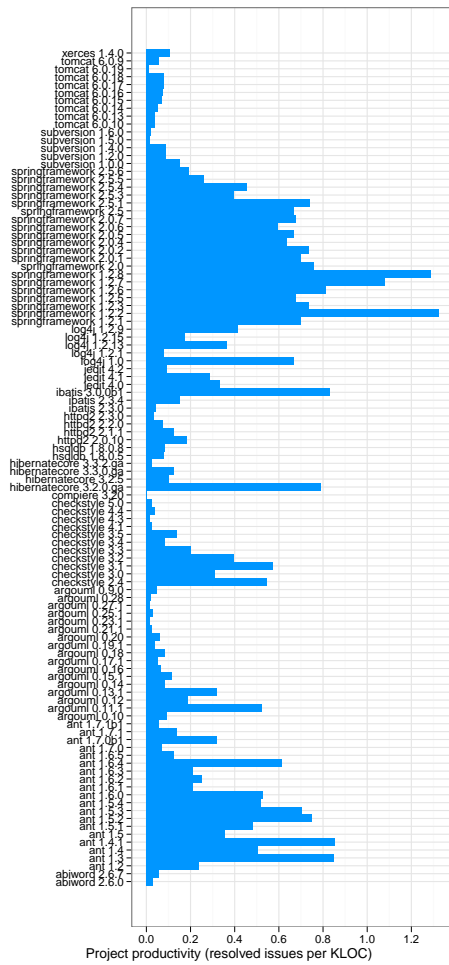


Figure 7.1: Project productivity.

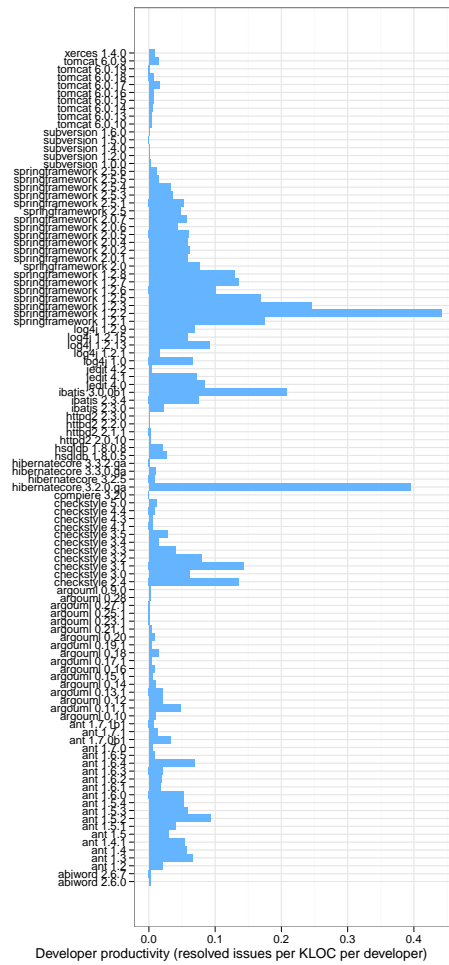


Figure 7.2: Developer productivity.

7.2.1 Quantification

Table 7.3 shows the median project productivity and developer productivity for each of the maintainability star ratings. As with the other indicators, there are no results for 1 and 5 stars because the dataset does not contain any snapshots with those ratings.

Like the results for the other indicators, the table also shows the ratio between the values and the value for average maintainability (3 stars). Developer productivity shows large differences between 2, 3, and 4 star systems. Project productivity shows a similar difference between 3 and 4 star systems, but the difference between 2 and 3 star systems is relatively minor. This difference is depicted in Figure 7.3.

Maintainability rating	Snapshots	Median project productivity	Median developer productivity
5 stars	0	N/A	N/A
4 stars	40	0.559 ($\times 6.0$)	0.058 ($\times 5.8$)
3 stars	52	0.093	0.010
2 stars	14	0.077 ($\div 1.2$)	0.002 ($\div 5.0$)
1 star	0	N/A	N/A

Table 7.3: Project productivity and developer productivity grouped per maintainability rating.

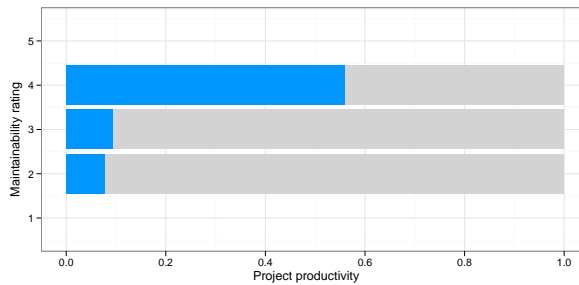


Figure 7.3: Median project productivity for each maintainability rating.

7.3 Discussion

These results show that systems of different maintainability ratings have different distributions in terms of both project productivity and developer productivity. For 2-star systems it appears to be very difficult to achieve high project productivity, as for most 3-star systems. A maintainability rating of 4 stars seems to be the point where there is a sharp increase in productivity.

Project productivity is measured by normalizing the number of resolved issues per month for system size. We use LOC as an indicator for system size, an alternative indicator would have been the number of issues in the ITS. An argument against using the number of issues as an indicator for system size is that the number of issues also depends on factors other than system size, such as the popularity of the system. LOC does not have these confounding factors. Also, using LOC to normalize for system size is used by well-known software quality metrics such as defect density [19].

It can be argued that the number of developers is also an indicator of system size, and by normalizing for both this and LOC we are in fact normalizing for system size twice. If that were true, project productivity would be a better indicator. Taking both LOC and developers into account can be defended by pointing out that the current number of developers may be

unrelated to how many developers have worked on the system in the past, and therefore that the current number of developers and LOC are indicators for different things. Also, if the number of developers and LOC would be highly correlated, normalizing by both would be the same as normalizing by LOC squared, which would have no influence on the rank correlation.

Another potential influence on the results for developer productivity is the way the number of developers is calculated. As was discussed, the method used to determine this number does not always give accurate results, although the results are more reliable than the alternative approach that was considered (using the *assignee* issue property).

The results show a large jump in project productivity and developer productivity between maintainability ratings of 3 and 4 stars. This is surprising, since the results for resolution time and enhancement ratio did not show such a jump. Apparently there is a factor that causes developers of systems with a 4-star rating to solve much more issues.

One possible reason for this could be that in these projects issues are picked up immediately, and as a result both maintainability and productivity are high. Looking at the data shows this is not true, many of the systems with 4-star ratings have over 5,000 open issues.

Another possible explanation is that systems with 4-star ratings are backed by companies, allowing for a number of developers that are working on the system fulltime. Looking at the data shows that this is partly true, there are 4-star systems that have commercial backing (Spring Framework), but there are also 4-star systems worked on by individuals (Checkstyle).

7.4 Conclusion

The results show a positive correlation between project productivity and all SIG Quality Model ratings, rejecting [H1]. A similar but slightly higher correlation is seen between developer productivity and all SIG Quality Model ratings. This confirms [H2].

The correlation for developer productivity exists for all three levels of the SIG Quality Model: maintainability, its sub-characteristics, and the source code properties. The correlation is stronger for higher levels of aggregation. This implies that the relation between maintainability on a higher level and developer productivity is more clear, but also exists for all of the source code properties that form the rating.

For project productivity the correlation also exists for all levels of the SIG

Quality Model. As with developer productivity, all results are significant. The effect of adding manpower to keep project productivity high can be observed in the lower correlations of project productivity compared to developer productivity. However, the influence is lower than expected since there is still a positive correlation. This means that [H1] was formulated too strongly, since the expected effect was seen in the results, only to a lesser extent.

Chapter 8

Conclusion

In this thesis, we have defined four different indicators of issue handling efficiency. One of these indicators was based in a previous study [25].

An empirical study found a positive correlation between all four indicators and the maintainability of the associated software system, determined by its rating in the SIG Quality Model.

A number of research questions were formed in Section 1.2. This section will summarize the answers to those questions.

[RQ1] Does the correlation between defect resolution time and the SIG Quality Model ratings still exist?

After defining what issues have a life cycle that is not consistent with typical ITS usage, and removing those issues from the dataset, the correlation with the SIG Quality Model ratings is still present. In fact, it is slightly stronger than in the preceding experiment.

[RQ1.1] How much shorter or longer are defect resolution times for each of the SIG maintainability ratings?

We quantified the resolution times by taking the median resolution time for each of the maintainability ratings. According to these results, solving a defect in a 2-star system is 2.3 slower compared to a 3-star system. Solving a defect in a 4-star system is 1.5 times faster than in a 3-star system.

[RQ2] Does the correlation also exist for issues of type *enhancement*? And if so, is it stronger?

Using the same dataset as for the experiment described in [RQ1], we found a positive correlation between issues of type *enhancement* and the SIG Quality Model ratings. This correlation was slightly weaker than for issues of type *defect*.

[RQ2.1] How much shorter or longer are enhancement resolution times for each of the SIG maintainability ratings?

Like defect resolution times, enhancement resolution times were grouped per maintainability rating. Compared to systems with a 3-star rating, resolution times for 2-star systems were found to be 3.1 times slower, and resolution times for 4-star systems 1.3 times faster.

[RQ3] What other indicators of issue handling efficiency can be defined?

[RQ3.1] Are these indicators correlated with the SIQ Quality Model ratings?

[RQ3.2] What are their values for each of the SIG maintainability ratings?

We defined three additional indicators of issue handling efficiency: enhancement ratio, project productivity, and developer productivity.

Enhancement ratio is the percentage of enhancements out of all resolved defects and enhancements. It is preferable to implement more enhancements, and fix less defects. A positive correlation was found between enhancement ratio and maintainability. A similar correlation was found for all sub-characteristics (*analyzability*, *changeability*, *testability*, and *stability*). Grouping per maintainability rating found that, compared to 3-star systems, 4-star systems have an enhancement ratio that is 13% higher, and 2-star systems have an enhancement ratio that is 13% lower.

Project productivity is the number of resolved issues per month, divided by LOC. Normalizing the number of resolved issues per LOC is necessary to compare the number of resolved issues between systems of different sizes. The found correlations between project productivity and the SIG Quality Model ratings is stronger than those between enhancement ratio and the ratings. The sub-characteristic *stability* has a lower correlation than the others. Quantifying project productivity found a small increase from a 2-star maintainability rating to a 3-star one, and a very large increase from that to a 4-star rating.

Developer productivity is project productivity divided by the number of developers that worked on a system in the observed time period. This

indicator can be used to compare productivity for a system regardless of how much manpower is invested. A correlation was found between developer productivity and all ratings in the SIG Quality Model. Also, for all ratings the correlation was stronger compared to project productivity.

8.1 Discussion

The indicators that were defined in this thesis can be used to measure issue handling efficiency from different perspectives.

Issue resolution time measures how quickly issues are resolved, which is an important metric in environments where fast response times are required.

Enhancement ratio measures the throughput of issue handling. A low enhancement ratio means that developers are not able to implement many enhancements, since they are spending their time fixing defects. Enhancements are user requests to change the system so that it becomes more useful to the reporter. Therefore, enhancements that are not implemented will eventually lead to the system becoming less useful [22].

As the names imply, project and developer productivity measure the productivity of issue handling. Productivity is usually defined as the output per unit. In the case of issue handling, output can be defined as the number of resolved issues. For project productivity, we define a unit as time. For developer productivity, we define a unit as a man month.

As explained in Section 1.1, we expected software maintainability to have an influence on issue handling efficiency. Although our results cannot be used to determine the causality, the fact that we found a correlation means that our experiments do not disprove our theory.

8.2 Threats to validity

There are several factors that could potentially weaken the validity of our results. Some of these have already been mentioned elsewhere in this thesis, but are still listed here for completeness.

The threats to validity are split into three categories (construct validity, internal validity, and external validity) as suggested by Perry et al [32].

8.2.1 Construct validity

Do the variables and hypotheses of our study accurately model the research questions?

Unable to determine issue life cycle Issue resolution time includes both the time an issue is open, and the time an issue is being worked on. It is not possible to make a distinction between these two phases because of the way ITSs are used in most projects. Although this means that the indicator is a combination of two different phases in an issue's life cycle, we still found a correlation with the SIG Quality Model.

Assumption of equal distribution of work To obtain developer productivity, we divide project productivity by the number of developers. This assumes that all developers have an equal workload, which might not be the case. In fact, according to literature there can be large differences in workload between project members [35].

8.2.2 Internal validity

Can changes in the dependent variables be safely attributed to changes in the independent variables?

Unequal number of snapshots per system Some systems in the dataset have only one snapshot, others have more than twenty. This means that systems with many snapshots have more influence on the results. It was attempted to counter the influence of individual systems by making sure that they are comparable, as was described in Chapter 5. If all systems in the dataset are comparable, it should be irrelevant if some systems have more influence on the results.

Method of determining number of developers (1) The number of developers is calculated by counting the number of authors of commit messages in the VCS. Some projects might use a system where only some developers have commit rights, and the other developers send patches. In this case the counted number of developers would be too low. Examining project rules has shown that this is not a rule for systems in the dataset. Although it is possible that there have been some cases where a patch was added to an issue as an attachment, this is unlikely to have influenced the results.

Method of determining number of developers (2) Two systems (Ant and Hibernate) switched VCS during the researched period, using a

migration tool to preserve their commit history. Although manual inspection did not reveal any problems, the possibility of mistakes being made by the migration tool cannot be excluded.

8.2.3 External validity

Can the study results be generalized to settings outside the study?

Generalization to commercial systems All systems in the dataset are developed in open source projects. In open source projects developers are usually not in the same location, which might mean they put more emphasis on written communication (such as ITS usage) compared to commercial projects, where developers are typically in the same location. However, a study [24] has shown that the quality of commit messages (another form of written communication) in open source projects is not higher than those in commercial projects. We therefore expect the same for the quality of ITS data.

Another reason for believing that our results can be extended to commercial projects are the many similarities in process between open source projects and commercial projects. Both use daily builds, unit testing, continuous integration, and (elements from) agile methodologies. Because of these similarities, we expect that our results can be generalized to commercial projects that use a process that contains these elements.

Generalization to large systems The systems in the dataset are comparable in size, roughly between 30 and 300 KLOC. It is unclear if our assumptions about ITS usage can be extended to (much) larger systems. Additional experiments are needed to determine if our indicators for issue handling can also be applied to larger systems.

Generalization to other programming languages The large majority of systems in the dataset use Java as the main programming language. It can be assumed that our results can be extended to systems written in languages with similar characteristics, but it is unclear if this is also true for programming languages with very different characteristics. Examples are PHP, a dynamically typed scripting language mostly used for web programming, and COBOL, a procedural language.

Generalization to projects that do not use ITSs All projects that were researched use an ITS to manage the reporting and resolving of issues. There are other ways to manage issue handling, an example of which

is todo-list software such as Basecamp¹. The characteristics of issue handling in these projects might be completely different from projects that use an ITS. Therefore, our results cannot be used to make assumptions about issue handling for these projects.

8.3 Future work

This section describes a number of possible ways in which this study could be extended. The suggestions are split into three categories: extending the dataset, further investigation, and additional indicators.

8.3.1 Extend the dataset

Add more systems More systems can be added to the dataset to strengthen the conclusion, assuming that the added systems are comparable to the ones that already in the dataset. A number of criteria for these systems were mentioned in Chapter 5.

Add snapshots with a maintainability rating of 1 and 5 stars We have quantified all indicators for maintainability ratings of 2, 3, and 4 stars. We were not able to do the same for maintainability ratings of 1 and 5 stars because the dataset did not contain snapshots with such a rating. Therefore, snapshots with these ratings should be added in order to quantify the full scale of maintainability ratings

Use more systems written in different languages Most systems in the dataset are mainly written in the Java programming language. In order to draw more general conclusions, the dataset can be extended with systems written in different languages. Systems written in languages comparable to Java (such as C#) will probably show similar issue handling, but very different languages require additional research.

8.3.2 Further investigation

Determine the influence of issue priority on resolution time It is possible that issues of low priority are handled differently. If the development team does not have the capacity to handle all incoming issues, it can be assumed that they will attempt to first solve the issues with higher priority. If this happens over a longer period of time, it could lead to a large amount of low-priority issues with long issue resolution

¹<http://basecamp.com>

times.

However, Bettenburg et al. [4] and Bird et al. [5] both conclude that the “real” priority of an issue is determined by several factors, and that the priority property is only one of these factors. Therefore, using only this property to determine the importance of an issue is not sufficient.

Future research should form criteria for high and low priority issues, based on more information than just the priority property. The two mentioned studies can be used to form these criteria.

Investigate different project productivity distributions Chapter 7 found a large jump between the median project productivities of snapshots with maintainability ratings of 3 and 4 stars. Some initial investigation of the underlying data shows that the distributions for 2 and 3 star snapshots are similar, but that the distribution of 4 star systems is very different. It should be investigated what kind of distribution

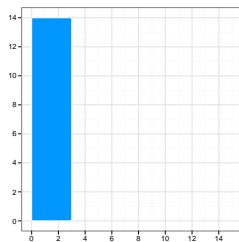


Figure 8.1: 2 stars

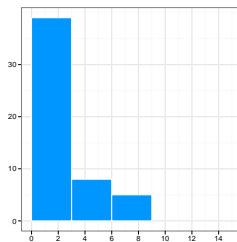


Figure 8.2: 3 stars

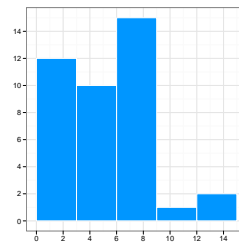


Figure 8.3: 4 stars

exist for each of the maintainability ratings, and what factors cause these distributions. This might lead to the conclusion that taking the median is not a good approach for quantifying project productivity, and that a different approach is needed.

Unit interfacing Regardless of which indicator for ITS usage is being used, the Unit Interfacing ratings consistently have the lowest correlation with these indicators. The reasons for this should be investigated. Are the ratings for Unit Interfacing different from ratings for the other source properties? Are these ratings consistently low or high?

Remove inactive issues The indicators described in this thesis look mainly at the number of resolved issues. If new indicators are going to look at open issues, the way we have cleaned the dataset may be inadequate. Issues that have been open for a very long time without any change made to them may never be fixed. Criteria for these inactive issues have to be created. For example, all issues that do not have status

changes for six months are considered inactive, and should therefore be removed from the dataset.

8.3.3 Additional indicators

Apart from the suggested indicators there are likely to be many more indicators of issue handling efficiency. Future work can identify more indicators, and compare them with the indicators defined in this thesis.

Percentage of backlog resolved One indicator that was considered for this thesis was the percentage of the issue backlog that was resolved in a certain time period. It was rejected for this thesis because it is unclear what factors influence the size of the backlog. If additional research identifies these factors, it might be possible to use this indicator.

Backlog management index (BMI) The book *Metrics and models in software quality engineering* [19] proposes the backlog management index (BMI) as an indicator for the efficiency of the maintainability process. This indicator is defined as

$$BMI = \frac{\text{resolved issues per month}}{\text{reported issues per month}} \times 100\%$$

A BMI of more than 100% means that the size of the backlog is being reduced. It will be interesting to see if this indicator, which is aimed at measuring the quality of the process, is related to the maintainability of the software.

Number of reopened issues In theory, software that is not maintainable is more likely to introduce new defects when fixing defects. Therefore, the number of issues that is reopened should have a strong correlation with the *stability* sub-characteristic of the SIG Quality Model.

Number of defects reported Instead of looking only at how many defects are resolved, it might also be interesting to look at how many are reported. In theory, high maintainability should lead to fewer defects. Unfortunately there are many factors other than maintainability that influence the number of reported defects, project popularity being the most important one. Therefore, the number of reported defects needs to be normalized before it can be compared between systems.

Bibliography

- [1] ALKHATIB, G. The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice* (1992).
- [2] ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A. D., AND MERLO, E. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.* 28, 10 (2002), 970–983.
- [3] ARTHUR, L. J. *Software evolution: The software maintenance challenge*. Wiley-Interscience, 1988.
- [4] BETTENBURG, N., JUST, S., SCHRÖTER, A., WEISS, C., PREMRAJ, R., AND ZIMMERMANN, T. What makes a good bug report? In *SIGSOFT FSE* (2008), M. J. Harrold and G. C. Murphy, Eds., ACM, pp. 308–318.
- [5] BIRD, C., BACHMANN, A., AUNE, E., DUFFY, J., BERNSTEIN, A., FILKOV, V., AND DEVANBU, P. T. Fair and balanced?: bias in bug-fix datasets. In *ESEC/SIGSOFT FSE* (2009), H. van Vliet and V. Issarny, Eds., ACM, pp. 121–130.
- [6] CANFORA, G., AND CERULO, L. Fine grained indexing of software repositories to support impact analysis. In *MSR* (2006), S. Diehl, H. Gall, and A. E. Hassan, Eds., ACM, pp. 105–111.
- [7] CANFORA, G., AND CIMITILE, A. Software maintenance. *Handbook of Software Engineering and Knowledge Engineering* (2000).
- [8] CONTE, S. D., DUNSMORE, H. E., AND SHEN, Y. E. *Software Engineering metrics and models*. Benjamin-Cummings, 1986.
- [9] D’AMBROS, M., AND LANZA, M. Bugcrawler: Visualizing evolving software systems. In *CSMR* (2007), R. L. Krikhaar, C. Verhoef, and G. A. D. Lucca, Eds., IEEE Computer Society, pp. 333–334.

- [10] D'AMBROS, M., LANZA, M., AND PINZGER, M. A bug's life: Visualizing a bug database. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2007).
- [11] FISCHER, M., PINZGER, M., AND GALL, H. Analyzing and relating bug report data for feature tracking. *Published by the IEEE Computer Society* (2003).
- [12] GRAMMEL, L., SCHACKMANN, H., AND LICHTER, H. Bugzillametrics: an adaptable tool for evaluating metric specifications on change requests. In *IWPSE* (2007), M. D. Penta and M. Lanza, Eds., ACM, pp. 35–38.
- [13] HEITLAGER, I., KUIPERS, T., AND VISSER, J. A practical model for measuring maintainability. In *QUATIC* (2007), R. J. Machado, F. B. e Abreu, and P. R. da Cunha, Eds., IEEE Computer Society, pp. 30–39.
- [14] HOOIMEIJER, P., AND WEIMER, W. Modeling bug report quality. In *ASE* (2007), R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds., ACM, pp. 34–43.
- [15] HUNT, A., AND THOMAS, D. *The pragmatic programmer*. Addison-Wesley Professional, 1999.
- [16] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *ISO/IEC Standard 14764-2006: Software Engineering Software Life Cycle Process Maintenance*, 2006.
- [17] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9126-1: Software Engineering - product quality - part 1: Quality model*, 2001.
- [18] KAGDI, H. H., COLLARD, M. L., AND MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance* 19, 2 (2007), 77–131.
- [19] KAN, S. H. *Metrics and models in Software Quality Engineering*. Addison-Wesley, 2003.
- [20] KIM, S., AND WHITEHEAD, E. J. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on mining software repositories* (2006).
- [21] KUIPERS, T., VISSER, J., AND DE VRIES, G. Monitoring the quality of outsourced software. In *Proc. Int. Workshop on Tools for Managing Globally Distributed Software Development* (2007).

- [22] LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* (1980).
- [23] LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. Characteristics of applications software maintenance. *Commun. ACM* 21, 6 (1978), 466–471.
- [24] LIPPOLIS, R. De wijzigingshistorie als informatiebron voor toekomstige wijzigingen. Master’s thesis, University of Amsterdam, 2008.
- [25] LUIJTEN, B. The influence of software maintainability on issue handling. Master’s thesis, Delft Technical University, 2009.
- [26] LUIJTEN, B., AND VISSER, J. Faster defect resolution with higher technical quality of software. In *4th International Workshop on Software Quality and Maintainability (SQM 2010), March 15, 2010, Madrid, Spain* (2010).
- [27] LUIJTEN, B., VISSER, J., AND ZAIDMAN, A. Assessment of issue handling efficiency. In *MSR* (2010), J. Whitehead and T. Zimmermann, Eds., IEEE, pp. 94–97.
- [28] MCCABE, T. J. A complexity measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320.
- [29] MCCONNELL, S. *Code Complete: A practical handbook for software construction (2nd edition)*. Microsoft Press, 2004.
- [30] MOCKUS, A., FIELDING, R. T., AND HERBSLEB, J. D. A case study of open source software development: the apache server. In *ICSE* (2000), pp. 263–272.
- [31] OPPEDIJK, F. R. Comparison of the sig maintainability model and the maintainability index. Master’s thesis, University of Amsterdam, 2008.
- [32] PERRY, D. E., PORTER, A. A., AND VOTTA, L. G. Empirical studies of software engineering: A roadmap. In *Proceedings of the conference on the future of Software Engineering* (2000).
- [33] PIGOSKI, T. M. *Practical software maintenance - best practices for managing your software*. Wiley, 1996.
- [34] SPEARMAN, C. Demonstration of formulae for true measurement of correlation. *The American Journal of Psychology* (1907).
- [35] STILLER, C., AND LEBLANC, C. *Project-Based Software Engineering*. Addison-Wesley, 2001.

- [36] VAN DEURSEN, A., AND KUIPERS, T. Source-based software risk assessment. In *ICSM (2003)*, IEEE Computer Society, pp. 385–388.
- [37] YING, A. Predicting source code changes by mining revision history. Master's thesis, University of British Columbia, 2003.
- [38] ZELLER, A. *Why programs fail*. Elsevier/Morgan Kaufman, 2006.