Software Qual J (2012) 20:265–285
DOI 10.1007/s11219-011-9140-0

Faster issue resolution with higher technical quality of software

Dennis Bijlsma · Miguel Alexandre Ferreira · Bart Luijten · Joost Visser

Published online: 2 May 2011

Abstract We performed an empirical study of the relation between technical quality of software products and the issue resolution performance of their maintainers. In particular, we tested the hypothesis that ratings for source code maintainability, as employed by the Software Improvement Group (SIG) quality model, are correlated with ratings for issue resolution speed. We tested the hypothesis for issues of type *defect* and of type *enhancement*. This study revealed that all but one of the metrics of the SIG quality model show a significant positive correlation with the resolution speed of defects, enhancements, or both.

Keywords Software defects · Defect resolution · Maintainability · Source code metrics · Rank correlation · Issue tracker mining

1 Introduction

The ISO/IEC 9126 international standard for software product quality (International Organization for Standardization 2001) defines three perspectives on product quality: internal quality, external quality, and quality-in-use. These perspectives correspond to

This article is the extension of a workshop paper (Luijten and Visser 2010). It replicates the study on *defects* with a larger code base and it adds a second study on *enhancements*.

D. Bijlsma
University of Amsterdam, Amsterdam, The Netherlands
e-mail: mail@dennisbijlsma.com

M. A. Ferreira (✉) · J. Visser
Software Improvement Group, Amsterdam, The Netherlands
e-mail: m.ferreira@sig.eu

J. Visser
e-mail: j.visser@sig.eu

B. Luijten
Delft University of Technology, Delft, The Netherlands
e-mail: bart@bluijten.nl

three distinct phases in the lifecycle of a software product. Internal product quality concerns the quality of the product as can be perceived already in the construction phase, by observation of the product independent of its operation. Static analysis techniques, such as source code metrics, are the instrument of choice to determine internal quality. External product quality concerns the quality as it can be observed in the testing phase, by dynamic analysis of the product's behaviour in a test environment. Quality-in-use concerns quality as perceived by its users (in a broad sense, e.g. including owners of the business processes that it supports) when the software product is operational.

For internal and external quality, the ISO/IEC 9126 standard provides a breakdown of the overall notion of quality into six main characteristics and a further breakdown into over 20 sub-characteristics. One of the six main characteristics is *maintainability*, which is further broken down into *analysability, changeability, stability* and *testability*.

To operationalise the quality definitions of ISO/IEC 9126, the Software Improvement Group (SIG) has developed a pragmatic measurement model that maps a selection of source code metrics to the *maintainability* characteristic and its sub-characteristics (Heitlager et al. 2007). Being based on static analysis, this model measures maintainability under the perspective of *internal* product quality. It can be applied to software products already in the construction phase, i.e. before entering the testing or production phases. The model is employed by SIG in its assessment and monitoring services (Deursen and Kuipers 2003; Kuipers et al. 2007). It also provides the basis of the software product certification service offered by SIG in collaboration with TÜV Informationstechnik (TÜViT), which results in the quality mark *TÜViT Trusted Product Maintainability* (Baggen et al. 2010).

In a good quality measurement model, each selected metric has a strong relationship with the particular quality characteristic that is measured with it. In the case of the SIG quality model, this means that the maintainability metrics as measured on the source code should have a strong relationship with the maintainability of the software product as experienced during maintenance activities. There is ample anecdotal evidence that suggests such a strong relation, based for example on the testimonials of software engineers and managers of numerous software products that have been evaluated with the SIG quality model in the context of software assessment, monitoring, and certification. In this paper, we report on an empirical study that investigates the relationship between maintainability and actually performed maintenance in a systematic manner.

In order to compare maintenance activities to our maintainability metrics, we need to quantify these activities in some manner. Properties of maintenance activities that one would like to quantify include their effectiveness (*Do defects get solved correctly?*) and their efficiency (*How much maintenance effort is invested?*). Unfortunately, reliable data regarding correctness and effort of changes is notoriously hard to come by, simply because they are usually not recorded.

We have looked to the data available in *issue tracking systems* (ITSs) to find a substitute for effort data. These trackers record for each issue when it was reported and when it was solved. This gives us an indication of issue resolution speed, which in turn we use as a substitute for issue resolution effort. By aggregation of resolution durations for individual issues during limited time-frames, we can construct an indicator of maintenance efficiency for the version of the product under maintenance in that time-frame.

Thus, the experiment we report on seeks to answer the following research question:

What is the relationship between software product quality as measured by the SIG quality model and the speed at which issues in these products are solved by the development / maintenance team?

We have conducted the experiment on data retrieved from the issue trackers and source code of a range of open source software products.

In Sect. 2, we explain our methods for data collection and analysis, including the SIG quality model and the quantification of issue resolution speed. In Sect. 3, we describe the systems on which we have conducted the experiment. In Sect. 4, we give the results of analysing these systems, including the various hypotheses that we are able to accept. In Sect. 5, we list and discuss the threats to validity of our experiment. In Sect. 6, we compare our work to related efforts. In Sect. 7, we conclude the paper with a discussion of the relevance of our results and avenues for future work.

2 Methods

An overview of the experiment process is given in Fig. 1. The process consists of analysis streams for issues (explained in Sects. 2.1 and 2.2) and for source code (Sect. 2.3). Issues are selected and grouped by source code snapshot date (as indicated by the calendar icon). The results of these streams are combined in a correlation analysis (Sect. 2.4). As we will explain in Sect. 3, the resolution types of issues of different types can be observed to follow different statistical distributions, which leads us to perform separate analyses for issues of type *defect* and of type *enhancement*.

2.1 Issue analysis

The process of reporting and resolving issues for a system during its development and/or maintenance is often handled through the use of an ITS. Examples include Bugzilla[1], Issuezilla[2], Jira[3] and the SourceForge Issue-Tracker[4]. Typically, an ITS can record for each issue its type (e.g. *defect, enhancement, patch, task*), its state (e.g. *new, assigned, resolved, closed*), the date of submission and of each state change, the submitter, any comments by others, and indications of severity and/or priority.

We constructed a Java tool that allows us to capture information from ITS repository dumps. The tool includes a generic data model that can store the needed data from different issue trackers in a unified fashion. The data model is optimised for post-mortem queries on large batches of issues, rather than for views and updates of the latest version of individual issues.

For the experiment reported in this paper, we measured *issue resolution time* defined as the time an issue is in an open state. Thus, we look at the time an issue is marked as being in the *new* or *assigned* state but not in the *closed* or *resolved* state. If an issue has been closed and then reopened, all open intervals count towards the issue resolution time, but the intervals in which the issue was closed do not. We take this as an indicator of the effort that was spent on solving the issue, for lack of availability of more accurate data.

Since we want to compare issue resolution times with maintainability measurements for particular versions of software products (snapshots at particular dates), we need to group issues by product version. For each version (i.e. snapshot date), we count as relevant issues

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

[2] A modified Bugzilla by CollabNet, http://www.open.collab.net.

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

[4] http://issue-tracker.sourceforge.net/.

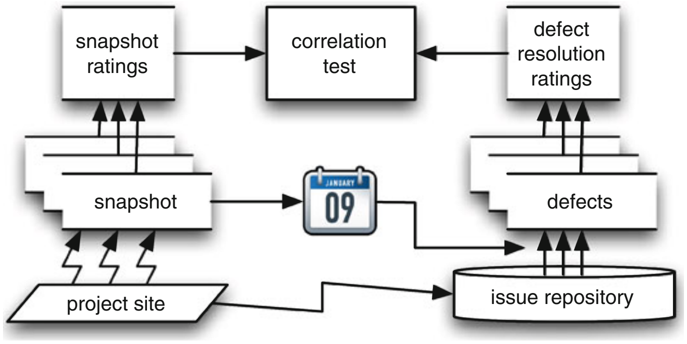


Fig. 1 Overview of the experiment

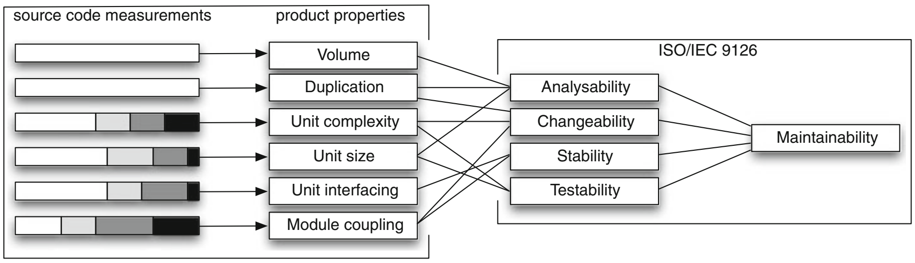


Fig. 2 The SIG Quality Model maps source code measurements onto ISO/IEC 9126 quality characteristics

those that are closed and/or resolved between that version and the next. We assume that most, if not all, of the work on solving an issue will be performed just before it is closed.

In this experiment, we perform separate analyses for issues of type *defect* and type *enhancement*. The processes of submission and resolution of issues of various types are characterised by different levels of urgency. Defects typically have high urgency. Enhancements typically have lower urgency or may even be regarded as non-essential. Tasks and patches are categories that unify issues of various types and urgency. Indeed, visualisation of issue churn for defects and other types bears out that they behave quite differently (Luijten 2010). Thus, pooling issues of different types does not seem to be justified.

2.2 Quantification of issue resolution efficiency

The resolution times of individual issues need to be aggregated to resolution efficiency ratings for the group of issues associated by date to each snapshot. In Fig. 3, the distribution of resolution times is visualised for various types of issues in a percentile plot. These plots show that issue resolution times do not follow a normal distribution, but rather a power-law-like distribution. Also, different types of issues follow slightly different distributions. For example, 80% of the defects are resolved within about 11 weeks (77 days), whilst 80% of the enhancements as well as 80% of the patches take almost one year to resolve (356 days).

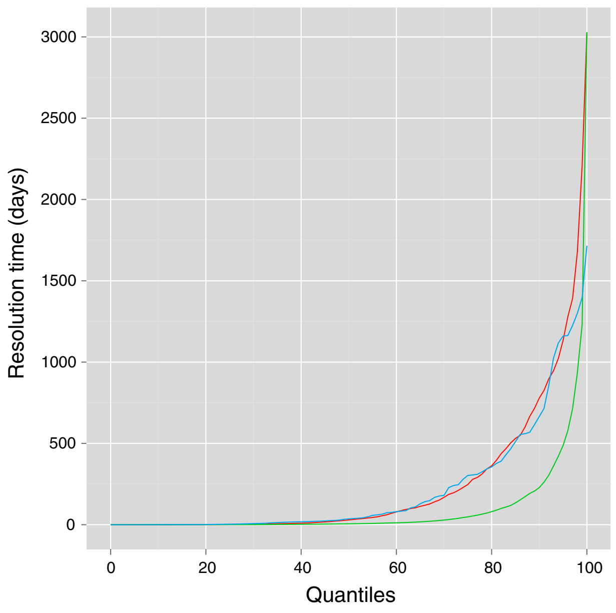


Fig. 3 Percentile plots of defects (*green*), enhancements (*red*), and patches (*blue*)

Based on these observations, the resolution times of different types of issues should not be pooled together but should be aggregated into separate ratings. Also, a simple aggregation by taking the *mean* or *median* of the defect resolution times is not appropriate for these asymmetric distributions. Instead we use so-called *risk profiles*.

A risk profile can be composed by assigning items to risk categories based on their metric values. For defect resolution time, we use the following risk categories:

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	

For example, a defect with a resolution time of 42 days falls into the *moderate* risk category.

Based on this risk assignment, a risk profile is constructed by calculating the percentage of items in each category. For example, $\langle 70, 19, 11, 0\rangle$ is the risk profile of a product history where 70% of all defects were resolved within 4 weeks, 89% were solved within 10 weeks, and none took longer than 6 months to solve.

Risk profiles can be mapped to ratings to enable straightforward comparison. We rate on a unitless scale between 0.5 and 5.5 that can be rounded to an integral number of *stars* (more is better). By benchmarking against defects for about 100 releases of various open source products (Luijten 2010), we calculated the following mapping:

Rating	Moderate (%)	High (%)	Very high (%)
*****	8.3	1.0	0.0
****	14	11	2.2
***	35	19	12
**	77	23	34

For example, a snapshot with defect risk profile $\langle 70, 19, 11, 0 \rangle$ will be eligible for a ranking of 3 stars. By interpolation our ranking algorithm establishes an exact rating of 3.25.

For issues of type *enhancements*, we performed the same procedure to construct the following risk categories and associated mapping to resolution ratings:

Category	Thresholds	
Low	0–152 days	(6 months)
Moderate	152–365 days	(1 year)
High	365–730 days	(2 years)
Very high	730 days or more	

Rating	Moderate (%)	High (%)	Very high (%)
*****	4.7	0.0	0.0
****	51	6.4	0.0
***	75	63	8.5
**	75	63	34

For example, a snapshot with enhancement risk profile $\langle 85, 9.5, 5.5, 0 \rangle$ will be eligible for a ranking of 4 stars.

We adopted the notions of risk profiles and star ratings from the SIG maintainability model (Heitlager et al. 2007) where they are used to aggregate source code metrics, rather than issues. This model is summarised below.

2.3 Source code analysis

The SIG has developed a layered model for measuring and rating the technical quality of a software system in terms of the quality characteristics of ISO/IEC 9126 (Heitlager et al. 2007). The layered structure of the model is illustrated in Fig. 2. In the first layer, source code analysis is performed to collect measurement data about the software system. The analysis involves well-known metrics such as LOC, duplicated LOC, McCabe complexity numbers, parameter counts and dependency counts. These metrics are collected on the level of basic building blocks such as lines, units (e.g. methods or functions) and modules (e.g. files or classes).

Subsequently, these metrics for building blocks are mapped onto ratings for properties at the level of the entire software product, such as volume, duplication and unit complexity. As in the case of defects, these ratings take values in the interval between 0.5 and 5.5, which can be rounded to an entire number of stars between one and five. This constitutes a unitless ordinal scale that facilitates communication and comparison of quality results at the level of entire software products.

The mapping functions for volume and duplication are straightforward translations of LOC-based metrics to ratings.

The remaining mapping functions make use of risk profiles as an intermediate device. Each profile is a partition of the volume of a system (measured typically by LOC) into four risk categories: low, moderate, high and very high risk. For example, if 4,000 LOC of a 100,000-LOC system sit in methods with a McCabe number higher than 50, the volume percentage in its very high risk category is 4.0%. Thresholds for metric values (like McCabe >50) have been chosen based on statistical study of a large set of representative systems. Likewise, thresholds have been chosen to define a mapping of risk profiles to quality ratings.

For the extraction of measurement values from source code, we rely on the SAT of the SIG. The SAT offers source code analysis capabilities for a wide range of languages as well as generic modules for aggregating and mapping source code metrics in accordance with the SIG quality model.

2.4 Statistical methods

The ratings for maintainability and for issue resolution time have an *ordinal* scale. Furthermore, both the maintainability ratings and defect resolution time do not follow the normal distribution in our sample. For these reasons, we have chosen to use the (non-parametric) Spearman rank-correlation method (Spearman 1907) for analysing correlation between these metrics, instead of the Pearson product-moment correlation coefficient.

3 Data

3.1 Software products used as subjects

The data we used for our experiments were obtained from the ITSs of 10 open source projects. See Table 1 for an overview. These projects were chosen because they are well-known projects and have publicly available source code and issue repositories. A few of the projects (notably ArgoUML and Checkstyle) were included because of the large amount of previous research done on these projects (e.g. Kim et al. 2006; Zaidman et al. 2008; Lubsen et al. 2009). For most of the systems, data have been collected for multiple snapshots, with a total of 120 snapshots. All systems have non-trivial size, with Checkstyle as one of the smaller systems (44 KLOC in the most recent snapshot) and Webkit as the largest (1.2 MLOC). The total number of registered issues for the 10 systems at the time of extraction was more than 61,000.

3.2 Data cleaning and selection

Several steps have been taken to reduce the data set for the experiment. An overview is presented in Table 2. Firstly, duplicate issues were removed, as well as issues that presented any kind of inconsistency. Also, issues were removed that could be established not to pertain to the source code. Secondly, snapshots with 5 or fewer issues of a particular type associated with them were rejected. From the resulting issues, we selected those with status *closed* or *resolved*. This resulted in a final set of about 11,000 defects in 89 snapshots, about 3.000 enhancements in 62 snapshots, and 264 patches in 23 snapshots.

Table 1 Software versions in dataset

Software product	Main language	LOC (latest)	Issue tracker	Snapshots (total)	Issues (total)
Abiword	C++	332,132	Bugzilla	3	10,941
Ant	Java	122,130	Bugzilla	20	5,192
ArgoUML	Java	171,402	Issuezilla	20	5,789
Checkstyle	Java	44,653	SourceForge	22	612
Hibernate-core	Java	145,482	Jira	5	4,009
JEdit	Java	54,235	Sourceforge	4	3,401
Spring-framework	Java	118,833	Jira	21	5,966
Subversion	C	218,611	Issuezilla	5	3,103
Tomcat 6.0	Java	163,589	Bugzilla	19	644
Webkit	C++	1,255,543	Bugzilla	1	22,016
$N = 10$				120	61,673

Table 2 Overview of data reduction

Software product	Snapshots with >5 defects	Defects	Snapshots with >5 enhancements	Enhancements	Snapshots with >5 patches	Patches
Abiword	2	312	2	23	0	0
Ant	16	1,856	15	712	0	0
ArgoUML	17	3,008	17	510	15	54
Checkstyle	17	419	0	0	0	0
Hibernate-core	4	553	4	347	4	64
JEdit	3	1747	0	0	0	0
Spring-framework	17	1,415	17	1,403	0	0
Subversion	4	635	5	179	4	146
Tomcat	8	272	1	11	0	0
Webkit	1	1447	1	37	0	0
N=10	89	11,664	62	3,222	23	264

We can observe that only three projects make use of issues of type *patch*. For this reason, we decided not to include patches in our analysis.

4 Results

For issues of type defect and of type enhancement, we tested the same hypotheses, which we present in Sect. 4.1. We then provide separate discussions of the results for each kind of issue in Sects. 4.2 and 4.3.

4.1 Tested hypotheses

We tested rank correlation between the calculated ratings for defect and enhancement resolution efficiency against each of the 11 ratings in the SIG quality model. Thus, for each

$x \in \{maintainability, analysability, changeability, stability, testability, ...\}$ and for each $t \in \{defect, enhancement\}$, we have the following null hypothesis and alternative hypothesis:

$H_0^{t,x}$: There is no relation between x and resolution efficiency for issues of type t.
$H_{alt}^{t,x}$: There is a relation between x and resolution efficiency for issues of type t.

As stated above, these hypotheses were tested using Spearman's rank correlation. Since we expect a positive correlation, we used a one-sided test.

4.2 Issues of type defect

4.2.1 Correlation results

The results of the rank-correlation test for issues of type *defect* are shown in Table 3.

All correlations, except for unit interfacing, are significant at a 99% confidence level. The significant correlation factors ρ_s vary between 0.29 and 0.62. This means that for all $x \in \{maintainability, analysability, changeability, stability, testability, volume, duplication, unitsize, unitcomplexity, modulecoupling\}$ we reject the null hypothesis $H_0^{x,defect}$ and maintain the alternative hypothesis $H_{alt}^{x,defect}$ that a relation exists with defect resolution efficiency. For x = *unit interfacing* we can not reject or accept either hypothesis. The remainder of this section discusses these results in more detail.

4.2.2 Correlation with system properties

The first six rows of the table concern the system properties. These are measured directly from the system source code. As stated, most of these correlate positively with the defect resolution rating.

The correlations to volume and duplication are lower (around 0.33) than the others. In a larger system, we expect it to take longer to locate the locations where fixes need to be made. In a system with more duplication, each defect might need to be fixed in multiple places. The lower correlation factors suggest that these effects of volume and duplication exist but have lower impact than the effect of the other properties in the model.

The correlations to unit complexity, unit size and module coupling are stronger (around 0.54). This demonstrates that, as expected, defects in systems with large and highly complex units (methods, functions, etc.) take more time to fix.

Table 3 Results of Spearman's rank-correlation test between ratings for defect resolution time and maintainability

Defect resolution vs.	ρ_s	*p*-value
Volume	0.33	0.001
Duplication	0.34	0.001
Unit size	0.53	0.000
Unit complexity	0.54	0.000
Unit interfacing	0.19	0.042
Module coupling	0.55	0.000
Analysability	0.57	0.000
Changeability	0.68	0.000
Stability	0.46	0.000
Testability	0.56	0.000
Maintainability	0.64	0.000

Also, systems composed of modules (files, classes, etc.) that have high inward coupling apparently take more time to fix. For unit interfacing (rated via a risk profile for the number of parameters of each unit), the correlation result is not statistically significant (p-value $>$ 0.01). Therefore, we can not accept or reject the associated hypotheses. It is possible that with a larger data set significant results can be obtained that either show absence or presence of correlation. Also, the unit interfacing rating could be related with a different measure of issue handling efficiency. These questions lie outside the scope of our experiment and will be discussed below as possibilities for future work.

4.2.3 Correlation to quality characteristics

The next four rows of the table concern the maintainability subcharacteristics that compose the intermediate level of the SIG quality model. The ratings of these subcharacteristics are calculated from the ratings of system properties. All four subcharacteristics correlate positively. Interestingly, the correlation factors tend to be higher than the ones for the system properties from which they are calculated. For example, the correlation factor for *testability* is 0.56, whilst the factors for unit size and unit complexity (from which the testability rating is calculated) are 0.53 and 0.54. Apparently, the mapping of properties to subcharacteristics reinforces the correlation.

The last row of the table concerns maintainability itself. Maintainability is correlated to defect resolution time with one of the highest factors of all (0.64), showing again the reinforcing effect of aggregation in the model.

4.3 Issues of type enhancement

4.3.1 Correlation results

The results of the rank-correlation test for issues of type *enhancement* are shown in Table 4. All correlations, except for duplication and unit interfacing, are significant at a 99% confidence level. The significant correlation factors ρ_s vary between 0.44 and 0.69. This means that for all $x \in \{maintainability, analysability, changeability, stability, testability, volume, unit\,size, unit\,complexity, module\,coupling\}$ we reject the null hypothesis $H_0^{x,enhancement}$ and maintain the alternative hypothesis $H_{alt}^{x,enhancement}$ that a relation exists with enhancement resolution efficiency. For $x \in \{duplication, unit\,interfacing\}$ we can not reject or accept either hypothesis.

In comparison with the correlation results for defect resolution ratings, we can observe that significance for the various tests are the same except for duplication. In case of defects, a significant correlation was registered for duplication, but for enhancements, the level of significance is not reached. For the results that present significance for both types of issues, the strength of the correlation is higher for enhancements in case of volume, module coupling and stability, but lower for the other tests.

We will now discuss these results in more detail as well as the differences with respect to the results for defects.

4.3.2 Correlation with system properties

The correlations of enhancement resolution speed to volume and module coupling are higher (above 0.60) than to the others. They are also higher than the corresponding correlations for issues of type *defect*. This difference in correlation results can be explained

Table 4 Results of Spearman's rank-correlation test between ratings for enhancement resolution time and maintainability

Enhancement resolution vs.	ρ_s	p-value
Volume	0.61	0.000
Duplication	0.02	0.448
Unit size	0.44	0.000
Unit complexity	0.48	0.000
Unit interfacing	0.10	0.213
Module coupling	0.69	0.000
Analysability	0.44	0.000
Changeability	0.46	0.000
Stability	0.50	0.000
Testability	0.47	0.000
Maintainability	0.53	0.000

from a difference in the locality of changes in case of defects and enhancements. Whilst defects typically require relatively local code changes, enhancements often involve larger refactorings and non-local changes. Both volume (of the entire system) and module coupling (to any of the other modules in the system) measure whole-system properties, whilst the other properties with significant results are aggregations of unit-level measurements.

The correlations to unit size and unit complexity are not as strong as those for volume and module coupling but can still be regarded as high (around 0.45). Thus, unit-level quality is an important factor, also for resolution of enhancements.

Unit interfacing does not present a statistically significant correlation to enhancement resolution speed, as was the case for defect resolution speed.

For duplication, the correlation result for enhancements is not statistically significant, as opposed to the correlation result for defects. This is possibly due to the number of enhancements in our data set (about 3,000), which is much lower than the number of defects (over 11,000).

4.3.3 Correlation to quality characteristics

The four maintainability subcharacteristics that compose the intermediate level of the SIG quality model all correlate positively with factors between 0.44 and 0.50. Also the overall characteristic of maintainability itself is positively correlated with enhancement resolution speed (0.53).

The correlation factors at the level of subcharacteristics do not tend to exceed the ones at the level of the properties from which they are calculated. Thus, at this level, we do not observe the same reinforcing effect of aggregation in the model that was observed in the case of defects. A possible explanation for this difference is that three of the subcharacteristics are calculated from a property for which no significant result was obtained (duplication in case of analysability and changeability and unit interfacing in case of stability). In the case of defects, this was only the case for stability (calculated from unit interfacing). Thus, the reinforcing effect of aggregation may be cancelled out by the insufficient alignment of the duplication ratings with the enhancement resolution ratings.

For maintainability, the reinforcing effect of aggregation can be observed in the case of enhancements, just as it was observed for defects.

If statistically significant results would be obtained for the duplication property, this could lead to the observation of a reinforcing effect at both levels, which would lift the correlation factors both for subcharacteristics and for the overall characteristic of maintainability. A future replication of this study with a larger number of enhancements is needed to validate this possibility.

5 Threats to validity

As suggested by Perry et al. (2000), we discuss three types of influences that might limit the validity of our study.

5.1 Construct validity

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

Lack of issue detail The data recorded in ITSs do not allow us to distinguish between different activities in solving an issue, such as *analysis* (finding out what changes need to be made), *modification* (actually changing the code) and *verification* (checking that the issue has indeed been resolved). Lacking such detail, we compare the *entire* issue lifetime to the *maintainability* characteristics, as well as its subcharacteristics (*analysability*, etc.) and the system properties (*complexity*, etc.). We expect this to *decrease* significance and strength of correlation, which means our positive results are conservative.

Idle time and double time Though we filter out the periods between closing and re-opening of an issue, the issue resolution time that we measure potentially includes idle time: though the issue is open, the team may not be working on it. On the other hand, our metric does not take into account that more than one team member may be working on the same issue at the same time. We believe that idle time is a common phenomenon and further study is warranted to discover whether it occurs in roughly the same degree in all issue types and in all projects. We believe that double time is rare and has a very limited effect on our study.

5.2 Internal validity

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

Unequal representation The dataset contains systems with just one snapshot and systems with multiple (up to 22). As a result, not all systems are equally represented and do not contribute with equal weight to the result. Still, multiple snapshots of the same system might be regarded as independent data points since they are separated in time and large variations can be observed amongst the ratings for different snapshots of the same system. Future experiments on larger datasets, or datasets without multiple versions of a single system, can counter this threat more definitively.

Confounding effects We did not measure some aspects of software projects that could have an influence on the issue handling process, such as team size or project popularity. Such factors could lead to larger or smaller numbers of issues being reported or resolved. We believe the influence of these factors to be small.

5.3 External validity

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

Bias due to issue tracker usage Our experiment includes only data from projects that make systematic use of an issue tracker. It is plausible this is a trait of well-organised projects that embrace best practises and follow a mature software engineering process. The established relationship between source code maintainability metrics and issue resolution efficiency should therefore not be assumed to hold also for projects at a lower maturity level.

Generalisation beyond Java Most projects in our study have Java as the main programming language. Even though the SIG quality model has been designed to include metrics that are applicable to a wide range of languages, the typical measurement values may be different per language. We believe that generalisation of the results of our study to languages with similar characteristics, such as C# and C++, is justifiable. For languages with different characteristics, such as COBOL (procedural), PHP (dynamically typed) or Erlang (functional), further experiments are warranted.

Generalisation to commercial projects We studied only data from open source projects. The organisation, management, and execution of open source projects typically differ from those of commercial projects. Still, the boundaries between these two kinds of projects are not strict. Many (successful) open source projects are initiated, supported, organised and/or managed by a commercial organisation (e.g. Oracle/SUN for OpenOffice.org, IBM for Eclipse, VMware/SpringSource for Spring Framework, Apple for Webkit, etc.). On the other hand, commercial software projects are increasingly adopting practises that have been common practise in open source projects for a longer period already, such as unit testing, continuous integration, usage of ITSs, globally distributed teams, nightly builds and (elements from) agile methodologies. Due to these blurring boundaries, we believe that our results can be generalised to commercial projects that are run in a "modern" way but not necessarily to classical waterfall-development projects.

6 Related work

6.1 The maintainability index

We start our related work discussion with a comparison to empirical studies conducted on the Maintainability Index (MI) (Oman et al. 1991), because the construction of the SIG maintainability model was originally motivated by criticism of the MI (Heitlager et al. 2007; Kuipers and Visser 2007).

Oman et al. (1991) introduced the MI as a source-code measure for maintainability. The MI is a four-metric polynomial, based on the Halstead effort metric, McCabe's cyclomatic complexity metric, lines of code and lines of comment. The constants in the formula that combines these metrics were initially established by a regression analysis against subjective assessments of maintainability for 8 software systems ranging between 1,000 and 10,000 lines of code written in C or Pascal (Oman and Hagemeister 1994). The subjective maintainability assessments of the subject systems were elicited from the system's maintainers by surveys based on the Air Force Operational Test and Evaluation Center (AFOTEC) evaluation guide (Air Force Operational and Evaluation Center 1989). All systems were programmed in Pascal or C and were obtained from two different sites of Hewlett-Packard. For the model obtained by regression analysis, a value of $R^2 = 0.90$ was reported as goodness of fit. A further 6 systems sized between 1,000 and 8,000 lines of code in the same languages were used to validate the model. A Spearman rank-correlation test was performed between the MI values calculated from the source code of these systems and the subjective assessments of their maintainability. A correlation factor was reported of $\rho = 0.74$, but *without statistical significance* ($p > 0.05$).

In spite of the lack of statistical significance, an improved version of the model was applied to 11 industrial software systems and their components where "[i]n each case, the results from [the] analysis corresponded to the maintenance engineers 'intuition' about the maintainability of the (sub)system components" (Coleman et al. 1994). The improvements of the model include using Halstead volume instead of Halstead effort and making it optional to use the metric for comment lines. On the original validation data, the average residual between the original and improved model is reported to be low (less than 1.4 for values ranging between 45 and 95).

The three-metric polynomial MI (i.e. without the optional factor for number of comment lines) was applied to all components of "a large industrial software system" and compared to the number of recorded defects for each of these 30 components (Coleman et al. 1995). A correlation factor is reported of 0.83, but no further details of this study were provided.

The three-metric and four-metric polynomial MIs were applied to various versions of the same software system (Welker et al. 1997). The earlier versions were developed in FORTRAN, whilst later re-engineered versions co-existed in C and Ada. For one version in each language, also a subjective assessment was performed. No statistical analysis of correlation was performed, but "[a]lthough the numerical values of the subjective assessments do not correspond exactly to the MI values obtained for those systems, the scale and direction of the results are similar" (Welker et al. 1997).

The MI and the SIG maintainability model show some overlap in their ingredient metrics, such as lines of code and cyclomatic complexity. However, the way these measurements are aggregated and combined are very different, with the MI taking the *mean* of these metrics whilst the SIG model constructs risk profiles. The two models also differ in the scale used for rating. In the SIG model, an ordinal scale between 0.5 and 5.5 is used on all levels, with a rounding convention to integral numbers between 1 and 5. The MI derives its scale from the 25-item AFOTEC evaluation guide where each item gets rated on an ordinal scale between 1 and 5, leading to an overall score between 25 and 125. The constructed polynomial model in practise attributes values on a wider scale, with values for entire systems reaching over 140 and values for parts of systems falling below −90.

Calibration of the MI model was done with 8 software systems of a single company written in procedural languages totalling no more than 80,000 lines of code. Calibration was done against subjective assessments of maintainability of these 8 systems by their maintainers. The SIG model is calibrated with systems of multiple organisation and open source communities with a total of more than 5 million lines of code written in modern languages such as Java and C#. Calibration is done against a desired symmetrical distribution of the subject systems over the rating scale.

The validation of the SIG model provided in the current paper can be compared with the validation reported by Coleman et al. (1995) that correlates the MI for system components with their recorded number of defects. Both validations use a defect metric as dependent variables, though we do not use defect counts for components, but defect resolution speed for the system as a whole. Our validation uses 120 snapshots of 10 open source systems, whilst the reported MI validation uses 30 components of a single proprietary system. In addition to a validation against defect data, we perform a validation against data for enhancements.

6.2 Prediction of defects

Most of the existing work that mines ITSs is focused on bug prediction. Examples of this are the works of Ratzinger et al. (2007) and Graves et al. (2000). Both attempt to predict the amount of bugs affecting a piece of software using the corresponding source and fault history. Interestingly, they identify the number of previous changes to a given module as a major influence, which is not a source code metric.

It is also possible to attempt to predict bugs in new or changed code by identifying the past changes that introduced a bug (Kim et al. 2006; Śliwerski et al. 2005; Kim et al. 2006; Aversano et al. 2007). This is done by examining the change log messages for indicators that a bug was fixed and assuming that the change that last touched the same code will have introduced the bug. Even though this seems to be a rather strong assumption, prediction accuracies of 35% (Kim et al. 2006) to 60% (Aversano et al. 2007) have been reported.

Gyimothy et al. (2005) study the Chidamber-Kemerer suite of object-oriented metrics (Chidamber-Kemerer 1994) within the Mozilla project[5]. They assign bugs in the ITS to source code in a specific release by examining patch files submitted with a bug. Using logistic regression analysis, they show that the Coupling Between Objects metric is the best predictor for fault-proneness, with precision and recall of just under 70%. However, due to the necessity of having patches available, their approach only takes a small percentage of bugs into account, which compromises generalisability (Bird et al. 2009).

Another evaluation of the Chidamber and Kemerer metrics was later done by Briand et al. (2000), who also examine a large number of metrics defined by others. They perform logistic regression techniques to determine impact on fault-proneness of the metric values. As a dataset, they use systems developed by students in a computer science course, with the faults determined by an independent team of professionals. Their best fitted model consists of four coupling and three inheritance measures, with a precision of 84% and recall of 94%. This result is consistent with others in placing emphasis on coupling metrics above size and complexity. Interestingly, the often-used McCabe complexity (1976) is not included in this analysis. A number of later works have replicated the verification of the Chidamber and Kemerer metrics, using various techniques such as threshold models (Benlarbi et al. 2000) and neural networks (Xu et al. 2008).

Zimmermann et al. (2009) examine the transferability of bug prediction models between projects. Out of the 622 project version combinations they tested, an astonishingly low 3.4% was able to reach their 75% cross-prediction accuracy threshold. Further investigation allows them to formulate a set of guidelines on how to select a project that will provide maximum prediction accuracy for a given target project. Unfortunately, Zimmermann et al. (2009) make no claims on the usability of a general model, trained on all subject systems.

A problem with some of the previously discussed work is the dependence on available links between VCS and ITS. In order to determine the source code impacted by an issue, many authors use issue identifiers present in commit logs, including (Kim et al. 2006; Ratzinger et al. 2007; Śliwerski et al. 2005; Kim et al. 2006; Aversano et al. 2007; Zimmermann et al. 2009; Fischer et al. 2003) . Both Ayari et al. (2007) and Bird et al. (2009) investigate the validity of this approach. Both show that a large amount of the issues present in the ITS are typically not traceable to source code. Bird et al. (2009) demonstrate that this leads to a bias in test results. This is a big threat to the validity of studies adopting this method. In our experiment, no use was made of VCS-ITS links. Rather, we looked at the total corpus of issues that are reported and closed for a product release or a time period.

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

Furthermore, we are not directly interested in predicting new issues but in the influence of software maintainability on the speed of issue solving.

6.3 Other maintainability indicators

Riaz et al. (2009) conducted a systematic review of studies into software maintainability prediction and metrics. They conclude that "there is little evidence on the effectiveness of software maintainability prediction techniques and models". In particular, out of 710 studies identified by automatic literature search, only 15 studies were deemed sufficiently promising for detailed assessment, which revealed that "metrics related to size, complexity and coupling were to date the most successful maintainability predictors employed". Four of the selected studies concern the maintainability index and were discussed above (Oman and Hagemeister 1994; Coleman et al. 1994, 1995; Welker et al. 1997).

Amongst the other studies identified by Riaz et al., several construct and test predictive models for maintainability based on various kinds of regression analysis (Zhou and Leung 2007; Misra 2005). Van Koten et al. (2006) use Bayesian Networks. Some of the identified studies use defect data as dependent variable, such as error rate (Ferneley 1999) and moments of detecting and correcting defects (Shibata and Okamura 2007).

Riaz et al. note that many of the identified studies rely on data-sets that limit generalisability of the results. This can be the case, for instance, when subject systems from a single source are used as in the various MI studies, when use is made of students to perform maintenance tasks, or when the number of data points is small. In our experiment, a large amount of source code from many different sources was used. Though all are open source systems, professional organisations are involved in their development.

7 Conclusion

7.1 Contributions

We have conducted an empirical study that demonstrated a significant, positive statistical correlation between the quality of software products as measured by the SIG quality model and the speed at which issues are solved by the development and/or maintenance teams of these products. The correlation was demonstrated separately for issues of type defect and of type enhancement.

The correlation was shown to exist on all three levels of the SIG quality model, i.e. at the level of *maintainability*, at the level of subcharacteristics of maintainability as defined by ISO/IEC 9126, and at the level of directly observable product properties, such as *volume, duplication* and *unit complexity*. For only one product property in the SIG quality model (*unit interfacing*, measured in terms of the *number of parameters per unit*), no correlation was found to resolution time of either defects or enhancements. For one product property (*duplication*), a correlation was found for defects, but not for enhancements.

The strength of correlation as found in the experiment regarding defects proved to increase at higher levels of aggregation. This suggests that the various metrics employed in the SIG quality model reinforce each other's effects. In other words, the quality model as a whole is more informative regarding issue resolution efficiency than any of its parts. In the experiment regarding enhancement, this effect was observed at the highest, but not at the intermediate level of aggregation, possibly due to the smaller size of the data set.

7.2 Discussion

The relationship between source code metrics for maintainability and duration of maintenance tasks established by our experiments constitutes an important empirical validation of the SIG quality model.

This validation was performed on the issues and source code of open source projects that use Java as the main programming language. Due to these characteristics, the study has immediate consequences for a large part of currently ongoing software engineering activities: issue resolution is one of the most important activities in software development and maintenance; recent years have seen a rapid growth of open source software as well as adoption of its best practises in commercial software projects; and Java is currently one of the most widely used programming languages.

Extrapolation of the experiment results beyond Java and beyond open source must be done with care. We have not identified *a priori* reasons why such extrapolation would not be possible. Nonetheless, an extension of the presented experiment to other programming languages and other development models would be of great added value. We explain our plans in that direction below (Sect. 7.3).

Extrapolation of the experiment results beyond issues of type defect or enhancement is not *a priori* expected to be possible, since other types of issues may have very different statistical properties. For example, we observed that the category of *patches* is used sparsely or not at all.

We believe that the experiment outcomes for the SIG model and its ingredient metrics can be understood to set a benchmark for other quality models and source code metrics. When proposing a new metric or new model intended to measure maintainability, a validation along the lines of the experiment described here can be performed to establish whether the proposal adds to the state-of-the-art. Also, when making modification to the SIG quality model, a repetition of the experiment should be carried out to determine the desirability of the modification.

7.3 Future work

Throughput ratings for other issue types In this experiment, we have focussed exclusively on issues of type *defect* and *enhancement*. It would be interesting to see whether it is also applicable to issues of other types. We have already identified *patches* as a possible issue type to study, but for this, a larger dataset is required.

Expand the dataset The dataset we used consists of ten open source projects, nine of which are represented by multiple versions. Increasing the size of this set will help in strengthening the confidence in the experiment results. In particular, it will be interesting to add commercial projects and to add projects with a main programming language different from Java. Also, with larger datasets, it is possible that significant (positive or negative) correlation results can be obtained for *unit interfacing* as well. In case of enhancements, a larger data set could yield significant correlation results for *duplication* that could possibly lift the correlation results at higher levels of the quality model through the reinforcing effects of aggregation.

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

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

Unit interfacing The lack of significant results for the *unit interfacing* system property gives rise to directions for further investigation. Repetition of the study with more data was mentioned above. Alternatively, a different source code metric could be chosen to quantify unit interfacing itself. In the current model, the number of parameters of units are used for this purposes. Alternatively, a weighted variant of this metric could be used (giving more weight to parameters of "richer" types) or a metric that takes not only explicitly declared parameters into account, but also access to variables. Finally, hypotheses can be formulated and tested that relate unit interfacing to other issue handling metrics (examples follow below).

Correlations with other indicators We quantified efficiency of maintenance activities in terms of issue resolution speed. Correlation of the SIG quality model with other indicators is of interest as well. An example is the number of people that work on issues. Also, the *effectiveness* of maintenance activities (*Are issues resolved correctly?*) is worth investigation. This could be measured for example by the number of times issues get reopened or by the number of new defects that are introduced whilst resolving an issue.

Acknowledgments Thanks to the developers of various open source systems for communication that helped us clean data and interpret results in our study.

References

Air Force Operational Test & Evaluation Center (AFOTEC). (1989). Software maintainability—evaluation guide. AFOTEC Pamphlet 800-2, Vol. 3, HQ AFOTEC, Kirtland Air Force Base, New Mexico, USA

Aversano, L., Cerulo, L., & Grosso, C. D. (2007). Learning from bug-introducing changes to prevent fault prone code. *IWPSE '07: Proceedings of 9th international workshop on principles of software evolution* (pp. 19–26).

Ayari, K., Meshkinfam, P., Antoniol, G., & Penta, M. D. (2007). Threats on building models from CVS and bugzilla repositories: The mozilla case study. *CASCON '07: Proceedings of 2007 conference of the center for advanced studies on collaborative research* (pp. 215–228).

Baggen, R., Schill, K., & Visser, J. (2010). Standardized code quality benchmarking for improving software maintainability. In *4th international workshop on software quality and maintainability (SQM 2010)*, March 15, 2010, Madrid, Spain.

Benlarbi S., Emam K.E., Goel N., & Rai S. (2000) Thresholds for object-oriented measures. *ISSRE '00: Proceedings 11th international symposium on software reliability engineering* (p. 24).

Bird, C., Bachmann, A., Aune, E., & Duffy, J. (2009). Fair and balanced?: Bias in bug-fix datasets. *ESEC/FSE '09: Proceedings of 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (pp. 121–130).

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

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

Coleman, D. M., Ash, D., Lowther, B., & Oman, P. W. (1994). Using metrics to evaluate software system maintainability. *IEEE Computer, 27*(8), 44–49.

Coleman, D., Lowther, B., & Oman, P. (1995). The application of software maintainability models in industrial software systems. *Journal of System Software, 29*(1), 3–16.

Deursen, A. V., & Kuipers, T. (2003). Source-based software risk assessment. In *Proceedings of international conference on software maintenance* (p. 385). IEEE Computer Society.

Ferneley, E. H. (1999). Design metrics as an aid to software maintenance: An empirical study. *Journal of Software Maintenance, 11*(1), 55–72.

Fischer, M., Pinzger, M., & Gall, H. (2003). Populating a release history database from version control and bug tracking systems. *ICSM '03: Proceedings of international conference on software maintenance 2003* (pp. 23–32).

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

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

Heitlager, I., Kuipers, T., & Visser, J. (2007). "A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)* (pp. 30–39). IEEE Computer Society.

International Organization for Standardization. (2001). ISO/IEC 9126-1: Software engineering-product quality-part 1: Quality model. Geneva, Switzerland.

Kim, S., Zimmermann, T., Pan, K., & Whitehead, Jr. E. J. (2006). Automatic identification of bug-introducing changes. *ASE '06: Proceedings of 21st IEEE/ACM international conference on automated software engineering* (pp. 81–90).

Kim, S., Pan, K., & Whitehead, Jr. E. J. (2006). Memories of bug fixes. *FSE '06: Proceedings of 14th ACM SIGSOFT international symposium on foundations of software engineering* (pp. 35–45).

Kuipers, T., & Visser, J. (2007). Maintainability index revisited—position paper. In *Special session on system quality and maintainability (SQM 2007) of the 11th European conference on software maintenance and reengineering (CSMR 2007)*, p. http://www.cs.vu.nl/csmr2007/workshops/SQM07_paper3.pdf .

Kuipers, T., Visser, J., & de Vries, G. (2007). Monitoring the quality of outsourced software. In J. van Hillegersberg, et al. (Eds.), *Proceedings of international workshop on tools for managing globally distributed software development (TOMAG 2007)*, Center for Telematics and Information Technology, Netherlands.

Lubsen, Z., Zaidman, A., & Pinzger, M. (2009). Using association rules to study the co-evolution of production & test code. *MSR '09: Proceedings of 6th international working conference on mining software repositories* (pp. 151–154).

Luijten, B. (2010). *The influence of software maintainability on issue handling*. Master's thesis, Delft University of Technology.

Luijten, B., & Visser, J. (2010). 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*.

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

Misra, S. C. (2005). Modeling design/coding factors that drive maintainability of software systems. *Software Quality Control, 13*(3):297–320.

Oman, P., & Hagemeister, J. (1994). Construction and testing of polynomials predicting software maintainability. *Journal of System Software, 24*(3), 251–266.

Oman, P. W., Hagemeister, J., & Ash, D. (1991). *A definition and taxonomy for software maintainability*. Software Engineering Test Laboratory, University of Idaho, Moscow, ID, USA, Tech. Rep. #91-08-TR.

Perry, D. E., Porter, A. A., & Votta, L. G. (2000). Empirical studies of software engineering: a roadmap. In *ICSE '00: Proceedings of the conference on the future of software engineering* (pp. 345–355). New York, NY, USA: ACM.

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

Riaz, M., Mendes, E., & Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *ESEM '09: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement* (pp. 367–377). Washington, DC, USA: IEEE Computer Society.

Shibata K., Rinsaka K., Dohi, T., & Okamura, H. (2007). Quantifying software maintainability based on a fault-detection/correction model. In *PRDC '07: Proceedings of the 13th Pacific rim international symposium on dependable computing* (pp. 35–42). Washington, DC, USA: IEEE Computer Society, 2007.

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). "When do changes induce fixes?" *MSR '05: Proceedings of 2005 international workshop on mining software repositories* (pp. 1–5).

Spearman, C. (1907). Demonstration of formulae for true measurement of correlation. *The American Journal of Psychology, 18*(2), 161–169.

van Koten, C., & Gray, A. R. (2006). An application of bayesian network for predicting object-oriented software maintainability. *Information & Software Technology, 48*(1), 59–67.

Welker, K. D., Oman, P. W., & Atkinson, G. G. (1997). Development and application of an automated source code maintainability index. *Journal of Software Maintenance, 9*(3), 127–159.

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

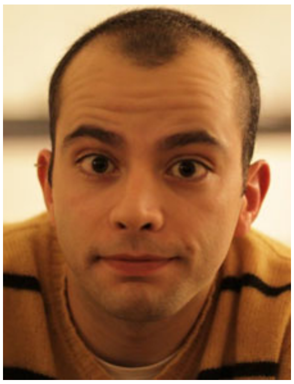
Zaidman, A., Rompaey, B., Demeyer, S., & van Deursen, A. (2008). "Mining software repositories to study co-evolution of production & test code," *ICST '08: Proceedings of 1st international conference on software testing, verification, and validation* (pp. 220–229).

Zhou, Y., & Leung, H. (2007). Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of System Software, 80*(8), 1349–1361.

Zimmermann, T., Nagappan, N., Gall, H., & Giger, E. (2009). Cross-project defect prediction: A large scale experiment on data versus domain versus process. *ESEC/FSE '09: Proceedings 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (pp. 91–100).

Author Biographies

Dennis Bijlsma holds a B.Eng. in Media Technology and a M.Sc. in Software Engineering. The latter was obtained at the University of Amsterdam in 2010, with a thesis exploring indicators of issue handling efficiency and their relation to software maintainability. The research for this thesis was performed in cooperation with the Software Improvement Group, which he has since joined as a software engineer.



Miguel Alexandre Ferreira obtained his M.Sc. degree from the Minho University, and has since then joined the Software Improvement Group as a researcher.

Bart Luijten obtained his M.Sc. degree from the Technical University of Delft during which period he was an intern at the Software Improvement Group. Since the completion of his M.Sc. studies, Bart has joined Mendix as a consultant.

Joost Visser is Head of Research at the Software Improvement Group (SIG) in Amsterdam, The Netherlands. In that role, Joost is responsible for innovation of tools and services, academic relations, internship coordination, and general research. Joost has worked for SIG before: in 2002–2003 he contributed to the initiation of Software Risk Assessment and Sofware Monitoring services. In the meantime, Joost was a researcher in the LMF Group at the Departamento de Informática of the Universidade do Minho in Braga, Portugal, and member of the Computer Science and Technology Center (CCTC). Joost carried out his PhD research at the Centre for Mathematics and Computer Science (CWI) in Amsterdam.