Eindhoven University of Technology

MASTER

Automating the reduction of risks associated with changes to software at SNS Bank N.V.

van Lümig, J.A.

Award date:
2015

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 28. Apr. 2025

Eindhoven University of Technology

Master's Thesis
Public Summary

Automating the reduction of risks associated with changes to software at SNS Bank N.V.

Arno van Lümig
arno@vanlumig.com

Supervisors:
Dr Alexander Serebrenik
Ir Wouter Poncin

Assessment committee:
Dr Alexander Serebrenik
Prof. Dr Mark G.J. van den Brand
Dr George H.L. Fletcher

March 2015

Abstract

With the increasing reliance on software to perform everyday tasks, software risk management is becoming a more and more important research topic. Banking software is no exception, so SNS BANK is interested in limiting the risks associated with changes to software. Since manual risk reduction techniques such as testing and code review are costly, automated tools may be used to reduce risks without significant cost increases.

We use mixed-methods research to determine what risks associated with changes to software are most important to SNS BANK. Based on this information, we develop a series of tools intended to reduce effort as well as increase quality.

One of these tools is an automated code review system, which integrates with the code review process that was already present. Findings are automatically inserted for certain problems that were discovered during the first part of our research.

Using interviews and experimental results, we validate that this code review system indeed contributes to the development process in a positive way. Based on this result, we make several recommendations to further develop the code review system.

Acknowledgements

First, I want to express my gratitude towards SNS BANK for giving me the opportunity to work on this project. In particular, I am grateful to Wouter Poncin, who has helped me navigate the organisation and the projects, and provided valuable feedback on this report. Many employees of SNS BANK have generously taken time out of their busy schedule to provide feedback and to ask the right questions, for which I am very thankful.

I could not have wished for a more patient, knowledgeable or passionate supervisor than Alexander Serebrenik. His guidance and advice has vastly improved the quality of this thesis. His extensive feedback during the six months of this project has been invaluable.

Last but not least, I could not have done this without the continued support of my parents and my sisters, who have provided me with everything I ever needed.

Contents

Chapter 1

Introduction

This document is the public version of my Master's thesis [27]. The full version of this thesis is not publicly available due to confidentiality agreements.

During recent years, banking software has become more and more complex. This is partly due to the introduction of new features such as electronic banking via websites and smartphones, but also due to changing regulations. For instance, the recent migration to IBAN and SEPA has required banks to quickly change existing functionality and implement new functionality. In 1996, Lehman claimed that "as an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" [24]. This claim is known as *Lehman's law*. Increasing complexity can lead to various problems such as bugs, difficulty implementing new features, testing a system becoming too difficult or labour-intensive, and technical debt [30].

These risks can be reduced using techniques such as automated and manual testing, code reviews, refactoring, and good work practices [4]. However, all these techniques have a high cost associated with them both in terms of money and in terms of time. As the ever-growing demand for new features and improvements continues, it becomes necessary to reduce these costs without causing an increase in these risks.

To reduce the risks without increasing the cost associated with these common risk-reduction techniques, SNS BANK is interested in developing an automated system that detects problematic changes as soon as possible.

By detecting problems earlier, they become faster and cheaper to fix [15]. By using automated tools, scarce human resources can instead be applied in more productive pursuits. This, in turn, will decrease the turnaround time on implementing new features and fixing issues.

Hence, the research goals for this thesis are as follows:

1. Identify which kinds of problems associated with changes to software are most important to SNS BANK;
2. Determine where automated tools should be applied to resolve these problems, or reduce the amount of effort required to resolve these problems;
3. Implement a tool or a set of tools;
4. Validate that this tool or set of tools indeed resolves some of the problems associated with changing software at SNS BANK.

1.1 Context

SNS BANK is a Dutch financial institution which is the parent company of several banks such as SNS, REGIO BANK and ASN BANK. SNS BANK is headquartered in Utrecht (the Netherlands). In total, approximately 6000 employees work at SNS BANK.

SNS BANK has hundreds of distinct software products which are developed in-house. The software development department at SNS BANK currently (July 2014) consists of approximately 470 employees.

While software development is not the core business of SNS BANK, software plays a crucial role in the business. For this reason, the reliability of the software is critical for the functioning of the business. Customers of SNS BANK expect their banking services to work correctly and reliably. Outages or bugs that are visible to end-users can significantly damage the brand image of SNS BANK.

My thesis was performed at the SNS BANK office in 's Hertogenbosch. We worked in the Electronic Banking team, which is part of the ITC V&O group (IT & change, veranderen ontwikkelbedrijf). V&O develops and maintains the IT systems that are necessary for the functioning of the bank, including back-end systems, the websites for the various labels, mobile applications, and various internal applications.

1.2 Methodology

Because this thesis is created in an industrial context, we have to choose a methodology that allows us to focus on providing value to SNS BANK. This is related to the philosophical stance that we adopt while performing this research, since the philosophical stance dictates the methodologies that can be used. Easterbrook et al. [8] provide an overview of philosophical stances as they apply to computer science.

Given the industrial context in which this thesis is developed, we take *pragmatism* approach during this research. The purpose of this research is to develop tooling that aids SNS BANK in developing software, and to achieve this we must do what is required. This naturally leads to a *pragmatism* approach. The pragmatism approach allows us to use qualitative methods such as interviews, surveys, and brainstorm sessions as well as empirical methods, all with the end goal of building tools that are useful to SNS BANK.

The *pragmatism* stance also influences how the developed tools can be validated. The tools are considered acceptable if (and only if) they are useful to SNS BANK. If the tools work in theory, but are not used in practice, then they have no value. This then implies that we must take human factors into account. For instance, we must ensure that the ease of use of the tools is such that they are indeed used by the right people, at the right time, and in the right way to help them achieve their goals more effectively

Chapter 2

Exploratory analysis

The first major part of the thesis was to determine what exact problems are faced by developers when making changes to software. Based on our pragmatist stance, we use mixed research methods which provides insight in both the current state-of-the-art in the literature as well as the software development as it happens at SNS BANK.

2.1 Literature study

A large amount of research has been done regarding risks such as bug-proneness [2, 3, 7, 10, 13, 16, 33, 39], maintenance difficulty [5, 9, 18, 20, 26, 34], reduced comprehension [14, 19, 23, 28, 36, 38], technical debt [6, 11, 12, 21, 22, 31, 37] and testing difficulty [29, 32]. This research provides inspiration on how to manage various risks that may arise when making changes.

Especially the aforementioned research area of bug-proneness has seen a large amount of research, much of it concentrating on various metrics. A study by Nagappan and Ball [33] compares various bug prediction approaches and is summarised in Table 2.1. From these results, it is clear that the models that are most predictive are also hardest to influence by automated tools.

Model	Precision	Recall
Organisational structure	86,2%	84,0%
Code churn	78,6%	79,9%
Code complexity	79,3%	66,0%
Social network/combination measures	76,9%	70,5%
Dependencies	74,4%	69,9%
Code coverage	83,8%	54,5%

Table 2.1: Results of the study by Nagappan and Ball [33]

2.2 Interviews

Using a semi-structured approach [17, 35] we interviewed five employees of SNS BANK that are involved in the software development process. These interviews took approximately half an hour each. While this is relatively short for an interview, it seemed quite possible to extract large amounts of information in this period without asking too much of the participants with respect to scheduling.

As prescribed by Seaman and Hove and Anda, the interviews were designed as follows: Interviews were conducted in a private room and recorded. Interviewees were assured that the notes and recordings made during the interview would only be used in anonymised form and only for this study. All interviewees consented to the recording. Before the interview the interviewer introduced himself and gave a very short summary of the purpose of the interview. Interviewees were encouraged to ask questions if something was unclear. Due to time constraints, it was chosen not to hold a trial interview, but instead leave some time after the first interview to transcribe and critically evaluate it. If modifications to the interview design were needed after this evaluation, the results of the first interview would be discarded.

Based on these interviews, we have found that post-release bugs are considered the worst risk associated with code changes. For this reason, we decide to focus the rest of this thesis on developing tools that improve the quality assurance process. More specifically, we focus on the *code review* process.

2.3 Review finding categorisation

All changes that are made to the systems we studied must first be reviewed by a team member. The reviewer will read through the *diff* of the changes made by the developer and look for problems. When the reviewer finds a questionable piece of code, a review finding is created that describes the problem, the location of the problem, and possible ways to fix it. Often, the reviewer will also ask questions to the developer regarding

design decisions and architecture. The developer can respond to review findings by fixing the issue, or by explaining why the finding is not an issue. When the review process is completed and all findings have been resolved, the changes can be applied to the main development branch.

Since we are interested in improving the code review process, we have manually categorised all these code review findings that were created since May 2011 into 75 categories. This has provided us with several ideas for tools that may be implemented.

2.4 Email survey

We sent an email survey to several employees to obtain more information on the code review process and possible areas of improvement. The email survey produced a number of findings that were not found during the review finding categorisation. These new findings are considered to be important by the email survey participants. We now have a reasonably complete list of ideas of where tooling may aid the review processes.

Many of the findings reported by the email survey participants were not present in large numbers in the review finding categorisation. This means that these findings are either infrequent, or hard to detect. If they are infrequent but considered to be important nonetheless, these findings probably have a large impact. If they are hard to detect, tooling may provide great benefit for these findings. In both cases should we take the reported findings seriously and further investigate them, even if the findings have not yet showed up in an earlier section.

2.5 Brainstorm session

The ideas generated have all been discussed in a brainstorm session and ranked according to their frequency and impact. With this, we have a complete idea of the exact problem statement as well as possible solutions. While there may in fact be other problems that are more important than the ones we have discussed so far, there is very little we can do at this point to find them.

The purpose of the brainstorm session was twofold. Primarily, we wanted to determine what kinds of issues that are associated with a change are considered most important by the session participants. Secondly, we wanted to obtain more background and explanation of some of the findings. This background information may be important for later design decisions, and when deciding how, when and to whom the tool output should be presented.

During the brainstorm session the participants openly discussed the various problems that they face and how important they consider them to be. By using a brainstorm

session we were able to eliminate misunderstandings that might have lead to wrong frequency or effort estimates.

For many of the findings the participants were able to provide detailed explanations of the findings and why they are (or are not) considered to be important. Based on these explanations, the participants were generally able to reach consensus on the frequency and impact of each finding.

Chapter 3

Tool development

We have developed eleven tools based on the results of the brainstorm session.

Presenting the output of these tools in a useful way is very important. Our pragmatic stance [8] dictates that our research should be conducted in such a way that it is the most useful to SNS BANK. According to this stance, a set of tools that is not used by the developers in practice has no value. For this reason we should try to integrate the tooling into the existing development process.

Most of the tools we implemented integrate with the code review process. When the changes made in a pull request introduce a problem that is detected by the tools, these tools will insert a finding into the code review process. This way, the result presentation is done in such a way that it has all three desirable characteristics described by Lewis et al. [25].

3.1 Test effort reduction

We have developed a relatively simple tool to estimate the number of modules that have changed between each release of two software systems. This tool is uses a dependency graph between modules to transitively detect changes. Testing effort can be reduced by only testing those modules that have changed.

Applying the developed tool to the releases of two large systems does not result in a considerable testing effort reduction. Developing more complex tools may produce more promising results, but will take considerable effort. We expect that the effort reduction will increase as projects moves towards continuous deployment, but this expectation cannot currently be validated.

3.2 Change coupling

Change coupling occurs when two artifacts often change at the same time [5, 26]. For instance, when a header file changes one expects the corresponding implementation to change also. In some cases, change coupling may indicate the presence of undesirable dependencies between code artefacts. For instance, when a piece of code is cloned then it may be the case that bug fixes applied to one instance of the clone are also applied to the other instances. If one instance is changed but the other is not, it may be useful to alert the developer of this fact.

By applying *association rule mining* [1] to files in a commit, one can detect change coupling between files. Additionally, by parsing the changed files in a commit it is also possible to extract *association rules* on smaller artefacts (e.g. classes or functions).

Change coupling has been shown to correlate to various issues that may exist in source code [5, 26], but this preliminary analysis has not produced promising results. Based on association rule mining on a file level, only 40 potentially useful rules were generated that only apply to a small number (2.3%) of commits.

3.3 Automated review

Nine of the tools have been developed to aid the reviewing process. The goals of these tools are as follows:

Reducing code review effort If common review findings are automatically detected, then the human reviewer does not have to create tickets for these findings.

Speed up code review By providing feedback to programmers sooner, the time required to fix the problems may be lower than if feedback had been provided later.

Improving code quality Some types of problems are easily overlooked during review. By automatically detecting these, code quality is improved.

Promote deeper code reviews One complaint about code reviews that we heard during the interviews was that reviews are sometimes too shallow. That is, some reviews only look for superficial problems. By automatically detecting some of these superficial findings, the human reviewer is encouraged to go more in-depth to find deeper problems.

To achieve these goals we integrated these tools into an automated code review system, that attempts to have the same external behaviour as a human reviewer. When a developer pushes a change to the repository, this change is automatically scanned for problems and these findings are inserted into the issue tracker similar to how a human would do this.

Chapter 4

Automated review evaluation

The automated code review system was developed and integrated into the standard development process. To validate the usefulness of the system, we enable it on two projects and gather data about the findings that are produced. Afterwards, we use semi-structured interviews [17, 35] with developers to further validate whether or not the tools provide value to SNS BANK.

4.1 Experimental evaluation

The review system was enabled for four weeks. The produced findings and they way in which they were resolved were stored in a database. All human-created review findings during the same period were also categorised according to the same categories that were used before.

Half of the automatically generated findings were closed as "Won't fix", especially those created by metrics-based tools (i.e. lines of code, cyclomatic complexity [29], and query cost). However, the other tools created much fewer findings. There were three "false positives".

4.2 Interviews

From the interviews, we conclude that the automated code review system is considered useful by most interviewees. There is a large degree if disagreement between interviewees regarding the usefulness of various tools. These differences can partially be explained by their functions and their previous experiences of problems that have occurred in the past.

Most interviewees are interested in adding more tools to the automated code review system, as well as improving the existing tools.

4.3 Conclusion

We have concluded that the system accomplishes several of its goals, while there is not enough evidence to draw conclusions on some other goals. Since the interviewees generally considered the system to be useful, we have nonetheless achieved our overarching goal of providing value to the team, and hence to SNS BANK.

Chapter 5

Future work

There are several known issues in the tools which could be addressed in future work. Some of these issues have been mentioned before, such as the test effort reduction tool that may benefit from a call-graph based approach.

Adding tools to the automated code review system may be one of the most important parts of future work. In this thesis, we have shown that the automated code review system is beneficial to the team, despite the fact that not all desirable tools have been implemented. Implementing these tools as well as discovering new tools that are beneficial to SNS BANK may significantly increase the value of the automated code review system.

Although we have performed some statistical validation by demonstrating that some found issues are eventually resolved, many issues were closed as "Won't fix" and there was not sufficient data to make strong claims. Statistically validating that the tools prevent bugs or improve code quality in the long run would be very beneficial as evidence for the usefulness of the tools.

Additional validation could be performed by applying these tools to more projects (within or outside SNS BANK), keeping in mind that not all tools may be useful to all projects.

Based on the experienced gained during this research, we make several recommendations to SNS BANK regarding further development of the tools. This includes implementing new automated review tools, running the tools periodically on the entire code base, extending the code review system to support different projects, and revisiting the *test effort reduction* tool once the transition to continuous deployment has been completed.

Chapter 6

Threats to validity

6.1 Construct validity

Given our pragmatic stance, validating this research happens based on the usefulness to SNS BANK. We have shown that the findings created by the code review tools were often missed by human reviewers, and we have shown that some of the developers who have used the automated review system found it useful. This alone is strong indication that the tools are useful to SNS BANK and hence the research is valid.

Although the results seem to imply that the tools are useful to SNS BANK, we cannot be sure that we implemented the tools that provide most benefit. For some tools, the false positive and false negative rates have not been analysed, and this may affect the validity of the tools.

6.2 Internal validity

We have shown that the automated code review system does indeed produce findings, some of which are resolved by the developer. Developers also closed some findings as "Won't fix", indicating that they are not afraid to "contradict" the tools. However, for example, they may only have resolved the findings because giving the reason for not resolving them (which is required when closing as "Won't fix") is more work than resolving them. If this would be the case, then we cannot claim the effectiveness of the automated code review system based on our obtained data.

However, we have used mixed methods research precisely because we cannot rely purely on the experimental data for reasons such as the one mentioned above. From the interviews, we did not get the impression that they resolved findings for any other reason than the fact that the findings were viewed as correct by the developers.

There are various threats to the internal validity of our research, but by using mixed-methods research we have reduced these threats somewhat. Since this research heavily relies on interviews and other human input, there are a few threats that are difficult to eliminate.

6.3 External validity

The external validity of the tools has not been determined, and it has been shown in the literature that bug prediction models are not transferable between projects. However, we believe that our approach of automatically adding findings to code review is valid in most projects that use a development process that includes code review.

Chapter 7

Conclusion

Software risk management is a complex and broad field of study. In this thesis, we have focused on a very narrow aspect of this field, namely detecting specific kinds of problems that may arise when a developer makes changes to code.

In the introduction of the thesis, we proposed the following research goals:

1. Identify which kinds of problems associated with changes to software are most important to SNS BANK;
2. Determine where automated tools should be applied to resolve these problems, or reduce the amount of effort required to resolve these problems;
3. Implement a tool or a set of tools;
4. Validate that this tool or set of tools indeed resolves some of the problems associated with changing software at SNS BANK.

We have achieved these goals as follows:

1. We have determined that the most important problem is that changes may lead to customer-facing bugs. Especially bugs that affect customer privacy or limit the access that customers have to their money are considered very important;
2. Automated tools can be applied during the code review process, since that is where a large amount of the risk-reduction effort goes. Additionally, applying automation at this stage in the development process is timely for both the developer as well as the reviewer. By integrating the tools with the existing development process there is very little added friction for the users of the tools;

3. We have implemented a set of nine automated code review tools, as well as two auxiliary tools that are not used in the code review system;

4. Several of the automated code review tools are considered useful by the participants, and do indeed detect problems in the source code. Some code review tools do not produce findings or the findings are not considered to be important. The two auxiliary tools have not produced promising results.

Bibliography

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining Association Rules Between Sets of Items in Large Databases". In: *SIGMOD Record* 22.2 (June 1993), pp. 207–216.

[2] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. "Fair and Balanced?: Bias in Bug-fix Datasets". In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2009, pp. 121–130.

[3] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. "Don't touch my code!: examining the effects of ownership on software quality". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM. 2011, pp. 4–14.

[4] Barry Boehm and Victor R. Basili. "Software defect reduction top 10 list". In: *Foundations of Empirical Software Engineering* 426 (2005), pp. 135–137.

[5] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "On the use of line co-change for identifying crosscutting concern code". In: *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. IEEE. 2006, pp. 213–222.

[6] Ward Cunningham. "The WyCash Portfolio Management System". In: *SIGPLAN OOPS Messenger* 4.2 (Dec. 1992), pp. 29–30.

[7] Marco D'Ambros, Michele Lanza, and Romain Robbes. "Evaluating defect prediction approaches: a benchmark and an extensive comparison". In: *Empirical Software Engineering* 17.4-5 (2012), pp. 531–577.

[8] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. "Selecting empirical methods for software engineering research". In: *Guide to advanced empirical software engineering*. Ed. by Forrest Shull, Janice Singer, and Dag I. K. Sjøberg. Springer, 2008, pp. 285–311.

[9] Stephen Eick, Todd Graves, Alan Karr, James Marron, and Audris Mockus. "Does code decay? Assessing the evidence from change management data". In: *IEEE Transactions on Software Engineering* 27.1 (Jan. 2001), pp. 1–12.

[10] Kalhed El Emam, Saïda Benlarbi, Nishith Goel, and Shesh Rai. "The confounding effect of class size on the validity of object-oriented metrics". In: *IEEE Transactions on Software Engineering* 27.7 (July 2001), pp. 630–650.

[11] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[12] Martin Fowler. *Technical Debt Quadrant*. Oct. 2009. URL: `http://martinfowler.com/bliki/TechnicalDebtQuadrant.html` (visited on 12/31/2014).

[13] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. "A systematic literature review on fault prediction performance in software engineering". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1276–1304.

[14] Maurice Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

[15] Bill Haskins, Jonette Stecklein, Brandon Dick, Gregory Moroney, Randy Lovell, and James Dabney. "Error Cost Escalation Through the Project Life Cycle". In: *INCOSE International Symposium* 14.1 (2004), pp. 1723–1737.

[16] Kim Herzig, Sascha Just, and Andreas Zeller. "It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 392–401.

[17] Siw Elisabeth Hove and Bente Anda. "Experiences from conducting semi-structured interviews in empirical software engineering research". In: *Proceedings of the 11th IEEE International Symposium on Software Metrics*. 2005, pp. 10–23.

[18] Magne Jørgensen. "Experience with the accuracy of software maintenance task effort prediction models". In: *IEEE Transactions on Software Engineering* 21.8 (Aug. 1995), pp. 674–681.

[19] Nadia Kasto and Jacqueline Whalley. "Measuring the difficulty of code comprehension tasks using software metrics". In: *Proceedings of the 15th Australasian Computer Education Conference*. 2013.

[20] Chris Kemerer and Knute Ream. *Empirical Research on Software Maintenance: 1981-1990*. Center for Information Systems Research, Sloan School of Management, Massachusetts Institute of Technology, 1992.

[21] Andrew Koenig. "Patterns and Antipatterns". In: *The patterns handbook: techniques, strategies, and applications*. 1998, pp. 383–390.

[22] Philippe Kruchten, Robert Nord, and Ipek Ozkaya. "Technical Debt: From Metaphor to Theory and Practice". In: *IEEE Software* 29.6 (Nov. 2012), pp. 18–21.

[23] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. "Effective identifier names for comprehension and memory". In: *Innovations in Systems and Software Engineering* 3.4 (2007), pp. 303–318.

[24] Meir M. Lehman. “On understanding laws, evolution, and conservation in the large-program life cycle”. In: *Journal of Systems and Software* 1 (1980), pp. 213–221.

[25] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and James Whitehead. “Does bug prediction support human developers? Findings from a Google case study”. In: *Proceedings of the 35th International Conference on Software Engineering*. 2013, pp. 372–381.

[26] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. “Evaluating the Harmfulness of Cloning: A Change Based Experiment”. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 18.

[27] Arno van Lumig. “Automating the reduction of risks associated with changes to software at SNS Bank N.V.” MSc thesis. Eindhoven University of Technology, 2015.

[28] Karl Mathias, James Cross, Dean Hendrix, and Larry Barowski. “The Role of Software Measures and Metrics in Studies of Program Comprehension”. In: *Proceedings of the 37th Annual Southeast Regional Conference*. ACM, 1999, p. 13.

[29] Thomas J. McCabe. “A complexity measure”. In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 308–320.

[30] Steve McConnell. *Technical Debt*. Nov. 2007. URL: `http://www.construx.com/10x_Software_Development/Technical_Debt/` (visited on 11/11/2014).

[31] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Françoise Le Meur. “DECOR: A method for the specification and detection of code and design smells”. In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36.

[32] Nachiappan Nagappan. “A software testing and reliability early warning (strew) metric suite”. http://www.lib.ncsu.edu/resolver/1840.16/4964. PhD thesis. 2005.

[33] Nachiappan Nagappan and Thomas Ball. “Evidence-Based Failure Prediction”. In: *Making Software: What Really Works and Why We Believe It*. Ed. by Andy Oram and Greg Wilson. O’Reilly Media, 2010, pp. 415–434.

[34] Paul Oman and Jack Hagemeister. “Metrics for assessing a software system’s maintainability”. In: *Proceedings of the Conference on Software Maintenance*. 1992, pp. 337–344.

[35] Carolyn Seaman. “Qualitative methods in empirical studies of software engineering”. In: *IEEE Transactions on Software Engineering* 25.4 (July 1999), pp. 557–572.

[36] Ted Tenny. “Program Readability: Procedures Versus Comments”. In: *IEEE Transactions on Software Engineering* 14.9 (Sept. 1988), pp. 1271–1279.

[37] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. “Design Pattern Detection Using Similarity Scoring”. In: *IEEE Transactions on Software Engineering* 32.11 (Nov. 2006), pp. 896–909.

[38] Anneliese Von Mayrhauser and A. Marie Vans. “Program comprehension during software maintenance and evolution”. In: *IEEE Computer* 28.8 (1995), pp. 44–55.

[39] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. “Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process”. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2009, pp. 91–100.