View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aaltodoc Publication Archive

Aalto University
School of Science
Degree programme of Computer Science and Engineering

Mikko Vestola

Evaluating and enhancing FindBugs to detect bugs from mature software: Case study in Valuatum

Master's Thesis
Espoo, May 10, 2012

Supervisor: Professor Casper Lassenius, Aalto University
Instructor: Juha Itkonen, D.Sc. (Tech.), Aalto University

Aalto University
School of Science

Aalto University, School of Science
Degree programme of CSE

ABSTRACT OF
MASTER'S THESIS

Author:	Mikko Vestola		
Title of thesis:	Evaluating and enhancing FindBugs to detect bugs from mature software: Case study in Valuatum		
Date:	May 10, 2012	**Pages:**	10 + 81
Professorship:	Software Engineering and Business	**Code:**	T-76
Supervisor:	Professor Casper Lassenius		
Instructor:	Juha Itkonen, D.Sc. (Tech.)		

Static code analysis (SCA) is a popular bug detection technique. However, several problems slow down the adoption of SCA. First of all, when first applying SCA to a mature software system, the SCA tools tend to report a large number of alerts which developers do not act on. Second, it is unclear how effective SCA is to find real defects. Therefore, we decided to conduct a case study in Valuatum to evaluate and enhance the effectiveness of FindBugs, a popular SCA tool for Java. The main goal of this thesis is to learn how to make FindBugs as an effective tool providing immediate, useful feedback for developers in Valuatum.

We have used several approaches to study FindBugs. First, we have analyzed how many and what types of fixed defects could have been prevented with FindBugs. Second, we have developed custom detectors for the most important defects missed by FindBugs. Third, we have studied the precision of FindBugs to detect open defects. Last, we have presented several approaches, such as defect differencing and IDE integration, to deal with the large number of alerts.

The results indicate that FindBugs is not very effective in detecting fixed defects. We estimated that 9–16% of the fixed bugs should be feasible to detect with SCA. However, only 0–2% of the reported fixed bugs and 1–6% of the unreported fixed bugs could have been prevented with FindBugs. Moreover, only 18.5% of the high-priority open alerts are considered actionable. Nevertheless, we think FindBugs is a cost-effective tool, because it detected several important open issues and can be enhanced with custom detectors.

Keywords:	FindBugs, Java, SCA, static analysis, static code analysis, static program analysis, static code checking, false positives
Language:	English

Aalto-yliopisto
Perustieteiden
korkeakoulu

Aalto-yliopisto, Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Mikko Vestola		
Työn nimi:			
FindBugsin arviointi ja tehostaminen kehittyneissä ohjelmistoissa: Tapaustutkimus Valuatumilla			
Päiväys:	10. toukokuuta 2012	**Sivumäärä:**	10 + 81
Professuuri:	Ohjelmistotuotanto ja -liiketoiminta	**Koodi:**	T-76
Työn valvoja:	Professori Casper Lassenius		
Työn ohjaaja:	TkT Juha Itkonen		

Staattinen koodianalyysi (SCA) on suosittu menetelmä ohjelmistovirheiden eli bugien etsinnässä. Sen käyttöönottoa kuitenkin haittaavat useat ongelmat. Ensinnäkin SCA tuottaa kehittyneissä ohjelmistoissa paljon varoituksia, joihin käyttäjät eivät reagoi. On myös epäselvää, kuinka tehokkaasti SCA löytää oikeita bugeja. Siksi toteutimmekin tässä työssä tapaustutkimuksen Valuatumilla, jossa arvioimme ja parannamme FindBugsin tehokkuutta. Työn päätarkoitus on oppia hyödyntämään FindBugsia niin, että se antaisi mahdollisimman hyödyllistä ja välitöntä palautetta Valuatumin ohjelmistokehittäjille.

Käytimme tutkimuksessa useita eri tapoja FindBugsin arviointiin. Ensinnäkin analysoimme, mitä korjattuja ohjelmistovirheitä FindBugs olisi voinut estää. Kirjoitimme myös omia bugi-ilmaisimia niille tärkeimmille virheille, joita FindBugs ei löytänyt. Lisäksi tutkimme FindBugsin löytämiä avoimia bugeja sekä esitimme erilaisia tapoja, miten hallita suuria määriä varoituksia.

Tutkimustulokset viittaavat siihen, että FindBugs ei ole kovin tehokas löytämään korjattuja ohjelmistovirheitä. Arvioimme, että SCA:lla pystyisi löytämään 9–16% korjatuista bugeista. Kuitenkin vain 0–2 % raportoiduista ja 1–6 % raportoimattomista bugeista olisi voitu estää FindBugsilla. Avoimista FindBugsin löytämistä korkeimman prioriteetin varoituksista vain 18,5 % luokiteltiin oleellisiksi. FindBugs on kuitenkin mielestämme kustannustehokas työkalu bugien etsintään, koska sitä pystyy tehostamaan omilla bugi-ilmaisimilla ja sen avulla olemme löytäneet useita tärkeitä virheitä koodista.

Avainsanat: FindBugs, Java, SCA, staattinen koodianalyysi, staattinen analyysi, väärät varoitukset
Kieli: englanti

Acknowledgements

I would first like to thank my instructor, Juha Itkonen, whose guidance and support during this project was very helpful. I am also grateful to my supervisor, Professor Casper Lassenius, who helped me to get started with the thesis and offered valuable comments and ideas.

Many thanks also go to my colleague and a good friend, Ville Saalo, who helped me with this thesis by providing encouraging feedback, valuable ideas, and support in proofreading. Furthermore, I offer my regards to my other colleague, Joni Rannila, who assisted me during the start of this project and enabled me to make this thesis to Valuatum.

Lastly, I would also like to show my gratitude to the original developers of FindBugs, William Pugh and David Hovemeyer, who have developed this useful tool for many years and share it to the software development community for free.

Espoo, May 10, 2012

Mikko Vestola

Abbreviations and Acronyms

AAIT	**Actionable alert identification technique**; A technique used to identify actionable alerts from the set of warnings produced by an SCA tool
AC	**Alert characteristic**; A characteristic of an alert from an SCA tool, usually used in AAITs
API	**Application programming interface**; Specifications that software programs can follow to communicate with each other
AST	**Abstract syntax tree**; A tree-form representation generated from program source code
CFG	**Control flow graph**; A representation of all paths that might be traversed through a program during its execution
CI	**Continuous integration**; A continuous process of applying quality control during software development
CPU	**Central processing unit**; A major component of a modern computer
CSE	**Computer Science and Engineering**
CSS	**Cascading Style Sheets**; A style sheet language describing the formatting of, for example, HTML pages
DFA	**Data flow analysis**; A high-level method for static code analysis
EJB	**Enterprise JavaBeans**; A server-side component architecture for modular construction of enterprise applications in Java

FN	**False negative**; A measure for instances where an SCA tool does not report a warning for a real defect
FP	**False positive**; A measure for instances where an SCA tool reports a warning which does not indicate a real bug
GPL	**GNU General Public License**; A free software license originally written for the GNU Project
GUI	**Graphical user interface**
HTML	**HyperText Markup Language**; A markup language describing the structure of web pages
I18N	**Internationalization**
IBM	**International Business Machines**; An American multinational technology and consulting corporation
IDE	**Integrated development environment**; A software application providing facilities to programmers for software development
JDK	**Java Development Kit**; A standard development kit for Java developers by Oracle Corporation
JSP	**JavaServer Pages**; A server-side Java programming language which is typically embedded to HTML pages
JVM	**Java virtual machine**; A virtual machine capable of executing Java bytecode
LGPL	**Lesser GNU Public License**; A free software license, more permitting than GPL
LOC	**Lines of code**; A unit used to measure software size
NASA	**National Aeronautics and Space Administration**; A space agency in the United States of America
NIST	**National Institute of Standards and Technology**; A non-regulatory federal technology agency in the United States of America
NCLOC	**Non-comment lines of code**; A software size measuring unit which ignores comments and line-breaks

NPE	**NullPointerException**; An exception which occurs when referencing a null object in program code
PMD	**Programming Mistake Detector**; A static code analysis tool for Java focusing on styling issues
QA	**Quality assurance**; A process used to measure and assure the quality of a product
SCA	**Static code analysis**; Program analysis without executing the analyzed program
SDK	**Software development kit**; A set of software development tools allowing the creation of applications for a certain software system
SPM	**Syntactic pattern matching**; A high-level method for static code analysis
SQE	**Software Quality Environment**; A code quality inspector plugin for NetBeans
SQL	**Structured Query Language**; A database query language
SVN	**Subversion**; A software versioning and a revision control system distributed by Apache Software Foundation
TN	**True negative**; A measure for instances where an SCA tool reports no warning because there is no defect
TP	**True positive**; A measure for instances where an SCA tool reports a warning which relates to a real bug
UI	**User interface**
URL	**Universal resource locator**; A reference to an Internet resource
XML	**Extensible Markup Language**; A markup language defining set of rules used to construct documents which are both human-readable and machine-readable
XSLT	**Extensible Stylesheet Language Transformations**; An XML-based declarative language used for the transformation of XML documents
XSS	**Cross-site scripting**; A type of security vulnerability typically found in web applications

Contents

Abbreviations and Acronyms **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Background 2

1.3 Research problem 3

1.4 Research questions 5

1.5 Structure of the thesis 6

2 Related work **7**

2.1 Methods for static code analysis 7

2.1.1 Syntactic pattern matching 8

2.1.2 Data flow analysis 8

2.1.3 Theorem proving 9

2.1.4 Model checking 9

2.2 Static analysis accuracy 10

2.2.1 False positives and actionable alerts 10

2.2.2 Actionable alert identification techniques 12

2.3 Evaluating the effectiveness of FindBugs 15

2.3.1 Detecting field defects 15

2.3.2 Measuring the false positive rate 17

2.3.3 Other effectiveness analysis 20

2.4 Enhancing FindBugs with custom detectors 21

3 Case study background **23**
3.1 Company and system background 23
3.2 Tool selection 24
3.2.1 Evaluated static code analysis tools for Java . . . 25
3.2.2 Reasons for selecting FindBugs 31

4 Case study methodology **34**
4.1 Study design: FindBugs effectiveness 35
4.1.1 Detecting reported fixed defects 35
4.1.2 Detecting unreported fixed defects 37
4.1.3 Detecting defects using custom detectors 39
4.1.4 Detecting defects based on alert removal history . 40
4.1.5 Detecting open defects 42
4.2 Study design: Dealing with unactionable alerts 43
4.3 Software used 44
4.4 Limitations 45

5 Case study results **47**
5.1 FindBugs effectiveness 47
5.1.1 Detecting reported fixed defects 47
5.1.2 Detecting unreported fixed defects 50
5.1.3 Detecting defects using custom detectors 53
5.1.4 Detecting defects based on alert removal history . 57
5.1.5 Detecting open defects 61
5.2 Enhancing FindBugs: Dealing with unactionable alerts . 64
5.2.1 Defect differencing 65
5.2.2 Integration to IDE 67
5.2.3 Simple AAITs 69

6 Conclusions **71**

7 Future work **73**

List of Tables

3.1 Summary of the main features of the SCA tools for Java . 31

5.1 Known reported bugs prevented with FindBugs. 48
5.2 Known reported bugs categorized as SCA feasible 48
5.3 Known unreported bugs prevented with FindBugs. 50
5.4 Known unreported bugs categorized as SCA feasible . . . 51
5.5 Distribution of removed warnings based on warning lifetime 58
5.6 Removed warnings classified based on BugRank 59
5.7 Removed warnings classified based on warning category 59
5.8 Open alerts having BugRank 1–9 classified based on alert category and importance level 62

List of Figures

5.1 An illustration of the deadlock with static synchronized methods 54

Chapter 1

Introduction

1.1 Motivation

A well-known fact in the field of software development is that fixing a defect early in the development cycle is much more cheaper than fixing the defect in the field. According to a NIST study published in 2002, software bugs cost about $60 billion annually merely for U.S economy (Research Triangle Institute, 2002, p. 169). Based on a feasible 50 percent reduction of errors through improved testing, the potential cost savings has been estimated as much as $22 billion. The potential savings are massive, so there clearly is a demand for cost-effective defect detection techniques.

Static code analysis (SCA) is one such technique which could help to save those 22 billion dollars. Nowadays, SCA is actively used in various software companies, such as NASA (Brat and Venet, 2005), Google (Ayewah and Pugh, 2010), IBM (Nanda et al., 2010), Microsoft (Ayewah et al., 2010), and eBay (Jaspan et al., 2007). Evidently, static code analysis is a very popular quality assurance method to find defects from software systems.

The main benefits of SCA are that it is well scalable and can be automated. Several studies have shown that SCA is a cost-effective fault detection technique (Baca et al., 2008; Jaspan et al., 2007; Wagner et al., 2008; Zheng et al., 2006). For example, Wagner et al. (2008) concluded that detecting a single severe defect or 3–15 normal defects is enough for an SCA tool to be cost-effective.

1.2 Background

Static analysis, also known as *static program analysis* (Godefroid et al., 2008), *static code checking* (Louridas, 2006), *automated static analysis* (Heckman and Williams, 2011) or simply *static analysis* (Ayewah, 2010), is defined by IEEE (1990) as:

> "The process of evaluating a system or component based on its form, structure, content, or documentation"

In practice, this means that SCA tools try to find out defects from the code without actually executing the program under analysis. Static code analysis is usually contrasted with dynamic analysis. In dynamic analysis, the program under analysis is executed by providing sufficient test inputs to produce interesting behavior, which can be further analyzed. Traditional testing methods, such as unit testing and performance testing, are examples of dynamic analysis (Emanuelsson and Nilsson, 2008). In contrast to dynamic analysis, static code analysis does not execute the analyzed program but it uses various formal analysis methods for defect detection, such as data flow analysis. SCA is not intended to replace traditional testing methods but is an additional quality assurance technique. It is especially useful for finding defects from untested code or detecting bugs which are hard to find with traditional testing, such as concurrency or security issues.

Numerous SCA tools are available and they differ, for example, from the formal methods they use and from the programming languages they support. In this study, we[1] focus on tools supporting Java programming language. These tools can detect defects such as null dereference errors, misuse of API, concurrency issues, cross-site scripting (XSS) vulnerabilities, and SQL injections. Some of the tools operate on source code, while others analyze bytecode. FindBugs, PMD, JLint, and Lint4j are examples of free SCA tools for Java. Also commercial products exist for Java, such as Klocwork Insight and Coverity Static Analysis.

A major problem with the commercial tools is that they are very expensive. The pricing policy seems to be mysterious: few companies offering commercial SCA tools list their price publicly. One article claims that

[1] Use of the plural pronoun is customary even in solely authored research papers; thus, we use the plural form also in this thesis.

Klocwork K7, the predecessor of Klocwork Insight, costs about $20,000 annually for projects up to a half-million lines of code (Louridas, 2006). Another problem with commercial SCA tools is that they usually come with license agreements forbidding the publication of any experimental or evaluative data (Ayewah et al., 2007; Hovemeyer and Pugh, 2007b). Free tools, however, have different problems: many of them are not actively developed anymore, so they suffer from, for example, incompatibility problems with the new versions of programming languages and out of date defect detection methods. FindBugs is actually the only free tool for Java which we saw worth analyzing (see more from Section 3.2). Therefore, the focus of this paper is on FindBugs.

The author of this thesis works in a software company named Valuatum. It develops a commercial Java-based system used by stockbrokers and their clients. A major problem in the system is that only 23% of the 700,000 lines of Java-code (including comments) are covered with automated unit tests. Much of the untested code is legacy code, which is hard to test with current unit testing tools. When most of the code is not automatically tested, even quite trivial programming errors might escape notice and end up to the production environment, causing visible problems to the customers.

In Valuatum, SCA has been seen as an economically potential method for detecting defects. It is not intended to replace unit testing but is an additional quality assurance technique to help detecting the most simple programming errors especially from the untested code. However, some well-known problems make the adoption of SCA challenging in Valuatum. These problems are discussed in the next section.

1.3 Research problem

One of the main problems slowing down the adoption of SCA in Valuatum is that **SCA tools tend to report a high number of false warnings** especially in mature systems which have not previously used any SCA tools. The rate of false warnings varies by tool, project, and configuration. A popular SCA tool, FindBugs, advertises on their website to have false warnings rate less than 50%. Heckman and Williams (2011) have reported that 35–91% of reported alerts are unactionable, in other words, developers do not react to these alerts. This high rate of false warnings may lead developers to completely ignore the output

of SCA tools because of the overhead of alert inspection. Based on a survey with the users of FindBugs, dealing with large number of false warnings is one of the biggest barriers for adopting SCA because it requires a significant initial effort the first time the tool is run (Ayewah and Pugh, 2008).

In fact, this is exactly what happened in Valuatum before starting this thesis. Although FindBugs was included in our development process for half a year, basically all the warnings from the tool were ignored by the developers. Running FindBugs to the whole Valuatum's system produced over 12,000 alerts. Many of the reported warnings seemed to be quite unimportant, and developers felt that it would take too much time to identify which warning is a real warning and which is a false warnings. There was clearly a need for a method or a process which could help to deal with the large number of alerts.

Another problem with FindBugs is that it is not clear **how effective it is to prevent actual defects**. For example, Wagner et al. (2008) evaluated the effectiveness of FindBugs. For one project, they did not find a single case where a field defect[2] was related to an alert reported by FindBugs. One reason for this outcome was seen that SCA tools are good at detecting commonly known general bugs, such as null dereferences, but defects occurring in the field are usually high-level logical project-specific defects. Another reason, pointed out by Plösch et al. (2008), is that the analyzed project was already in production for four years; therefore, reaching a high level of maturity.

Regarding the maturity level and project size, the larger project investigated by Wagner et al. is quite close to the software developed in Valuatum. Therefore, the results are disappointing from FindBugs' point of view and raise a couple of questions. First of all, is FindBugs really effective for detecting field defects from mature software? Second, if the field defects are usually project-specific, can we effectively customize FindBugs to find these project-specific bugs?

FindBugs allows users to develop own custom bug detectors, which could be used to detect at least some of the project-specific defects. Because all software contains internal APIs and have specific coding rules, these custom bug detectors could be used, for example, to find violations of project-specific coding rules or misuse of internal API. Several major companies have found out that developing own bug checkers

[2] The term "field defect" here means a defect which has been detected from the production environment, for example, by a customer.

might be cost-effective. For example, Jaspan et al. (2007) from eBay report that they are very interested in using own checkers to look for project-specific issues. Also Nanda et al. (2010) from IBM and Kienle et al. (2011) from ABB Robotics share the same interest.

In Valuatum, FindBugs has failed to identify several critical bugs which have caused considerable problems in the production environment. Some of these bugs were quite simple project-specific programming errors; therefore, they should be found with static code analysis. Although the aforementioned bugs are already fixed, SCA might help to prevent these bugs from occurring again. For that reason, using custom detectors in Valuatum seems to be an interesting approach and worth further studying.

The main goal of this thesis is to learn how to make FindBugs as an effective tool which could provide immediate, useful feedback for developers in Valuatum. To learn how we should use FindBugs effectively, we first need to analyze what FindBugs is capable of when using it in a mature system. How many and what types of defects does it actually find? Could some important missed defects be detected effectively with custom detectors? How many of the reported open alerts are actionable? All these questions need to be answered so that we can get the most out of FindBugs. In addition to evaluating the effectiveness of FindBugs, we also need to study how to deal with the large number of unactionable alerts.

1.4 Research questions

To answer the problems described in the previous section, this thesis studies the following research questions:

1. **RQ1**: How effective is FindBugs in preventing bugs in mature software systems?
2. **RQ2**: What techniques are applicable in mature software systems to find the most important alerts from the large number of alerts reported by FindBugs?

The research questions above are answered based on a case study performed in Valuatum. In summary, the main contributions of this paper are:

- Analyzing the effectiveness of FindBugs to prevent bugs in a mature software system. If some of the known bugs are not detected with the default detectors of FindBugs, we develop custom project-specific detectors for the most important missed defects. These custom detectors help us to prevent the defects from occurring again, and they might also reveal other locations of code infected with the same defect. Some of the custom detectors are generalizable; therefore, they may also help others to find the same defects in different projects.

- Analyzing the effectiveness of FindBugs to find new defects from a mature system by analyzing how many of the reported open alerts are actual defects having functional impact and how many are unactionable alerts.

- Presenting and evaluating different ways to deal with the large number of unactionable alerts in mature software systems. This will hopefully help also others to integrate FindBugs to their software development process.

1.5 Structure of the thesis

The rest of this thesis is organized as the following. In Chapter 2, we introduce related work about static code analysis. For example, different methods for static analysis are presented, and also previous studies about evaluating the effectiveness of FindBugs are described. Next, Chapter 3 presents some background information about our case study and why we have chosen FindBugs as the SCA tool for our work. To continue, Chapter 4 describes the case study methodology, in other words, how did we perform the study and what limitations are involved to the methods we have used. The actual results from our case study are presented in Chapter 5. Finally, in Chapter 6, we conclude the thesis and present some ideas for future work in Chapter 7.

Chapter 2

Related work

Static code analysis has been a quite active field of study. Numerous papers are published about the topic. In this chapter, we present the most important previous work related to the scope of this thesis. We first describe the methods for static code analysis ranging from simple techniques (such as syntactic pattern matching) to more complex methods (such as model checking). Second, we present the problem with false positives and what techniques exist for actionable alert identification. Third, we describe previous work on analyzing the effectiveness of FindBugs. Finally, related work on enhancing FindBugs with custom detectors is presented.

2.1 Methods for static code analysis

Several different methods exist for performing static code analysis. In this section, we present the most commonly used static analysis methods described in the literature. We include the following techniques mentioned in the taxonomy of static code analysis tools (Novak et al., 2010): *syntactic pattern matching*, *data flow analysis*, *theorem proving*, and *model checking*. Each of these methods are briefly described in the following subsections.

2.1.1 Syntactic pattern matching

Syntactic pattern matching (SPM) is one of the most common techniques used in static code analysis tools. FindBugs, PMD, JLint, and CheckStyle are examples of tools implementing SPM (Rutar et al., 2004; Novak et al., 2010). In practice, syntactic bug pattern detection scans the program code for suspicious patterns of code. Some of the tools (such as PMD) work on the source code and some (e.g. FindBugs) analyze the bytecode. The actual techniques used in bug pattern matching varies by tool. For example, PMD uses Java and XPath to detect bug patterns from the abstract syntax tree (AST) generated from the source code. FindBugs, on the other hand, linearly scans the bytecode of a class instruction at a time and, in that way, constructs a state for the class under analysis, which can be used to detect bug patterns.

One of the advantages of SPM is that it is rather easy to apply. Perhaps that is why the technique is one of the most popular static code analysis techniques used: almost all popular tools include SPM techniques. For many simple bugs, this technique is enough to detect them. Syntactic pattern matching is, however, said to be a quite shallow method and produces many false warnings (Emanuelsson and Nilsson, 2008). Therefore, it does not suite very well for more complex bugs which need more knowledge of the state of a program and interaction between different components (e.g. detecting more complex interprocedural null dereferences or finding complex synchronization problems).

2.1.2 Data flow analysis

Data flow analysis (DFA) is a technique used in more complicated static analysis bug detectors. It is perhaps the second most used static analysis technique after syntactic pattern matching. JLint and FindBugs are examples of tools implementing DFA (Rutar et al., 2004; Novak et al., 2010). Data flow analysis uses program's control flow graph (CFG) to analyze the application's execution paths. The goal of this method is to gather information about the possible set of values calculated at different points of program code. With the help of CFG, data flow analysis simulates application's execution paths without actually executing the code. Some tools might perform deep interprocedural data flow analysis, while others might be restricted to more lightweight intraprocedural analysis.

For instance, FindBugs has detectors which use intraprocedural DFA to detect null pointer dereferences (Hovemeyer and Pugh, 2004). Below is an example NPE (NullPointerException) bug which can be detected with FindBugs.

```
String s= getString();
if (s == null && s.isEmpty()) {
    return;
}
```

When compared to SPM, one advantage of DFA is that it can be used to detect more complex bugs than syntactic pattern matching. Many bugs require the knowledge of the possible values of variables. For example, good interprocedural null pointer dereference analysis requires knowing the possible set of values in order to determine whether a value can be null at some point of the program code.

2.1.3 Theorem proving

Theorem proving is a static analysis method which performs formal verification of the properties of program code. Theorem proving is used, for example, in ESC/Java2 (Flanagan et al., 2002; Rutar et al., 2004). In practice, ESC/Java2 provides programmers an annotation language which they can use to express design decisions formally, for example, by adding preconditions and postconditions. ESC/Java2 analyzes the annotated program code and warns of inconsistencies between the design decisions recorded as annotations and the actual program code.

One disadvantage of this method is that it has very poor tool support for Java: the discontinued tool ESC/Java2 is the only tool we are aware of supporting this technique. Another weakness of this technique is that the technique requires quite much manual pre-work, like adding annotations to the program code to record the design decisions in case of ESC/Java2. In large code bases, this manual pre-work might be too laborious.

2.1.4 Model checking

Another formal static analysis technique is model checking. It is used to extract finite-state models from program code. The generated model

can be then used to verify program behavior. This technique is used in the Bandera tool. To use Bandera, the programmers annotate their source code with specifications describing what should be checked, and with the help of the generated model, Bandera can verify the program behavior (Corbett et al., 2000).

Model checking has at least the same disadvantages as theorem proving: it requires quite much manual pre-work and has poor tool support for Java. Bandera is the only general-purpose SCA tool for Java we are aware of supporting model checking. Moreover, model checking might not be very well suitable for large programs because the state explosion problem makes it difficult to construct a finite state for large software systems (Corbett et al., 2000).

2.2 Static analysis accuracy

Regardless of the SCA technique used, almost all static analysis tools produce false warnings or miss bugs which they should warn of. Both of these are well-known problems with static analysis. In this section, we describe these problems in more detail and present some approaches how to deal with large number of false warnings.

2.2.1 False positives and actionable alerts

If a static analysis tool produces a warning which does not indicate a real defect, this warning is considered to be a *false positive (FP)*. Warnings revealing real errors are, on the contrary, *true positives (TP)*. Sometimes also terms *false negative (FN)* and *true negative (TN)* are used (Plösch et al., 2009; Heckman and Williams, 2008). The term false negative refers to unwanted cases where a tool does not report a warning for a real defect. True negative is a less commonly used term which means a desired case where a tool reports no warning because there is no defect.

False positives are probably the greatest problem with static analysis. If a tool reports a large number of false positives, developers will eventually lose faith to the tool and stop using it. This happened, for example, in eBay (Jaspan et al., 2007). It is clear that *effective* static analysis tools always produce false positives. The tools usually have to make a compromise between false negatives and false positives. In

the literature (Hovemeyer, 2005) this trade-off between false negatives and false positives is defined as terms *sound* and *complete* analysis. Sound analysis generally means that the tool "finds every real bug", whereas complete analysis "reports only real bugs". In other words, a sound analysis does not have false negatives, and a complete analysis does not have false positives.

Although completeness and soundness are desired, both of them are practically impossible to achieve in static analysis. If a tool can find all possible bugs (soundness) it usually does this by reporting hundreds or thousands of false positives for every true positive, making the tool unusable in practice. Instead, if the tool only reports true errors (completeness), it usually finds only a small number of defects, thus producing many false negatives.

The terms *precision* and *recall* are commonly used measures to analyze the effectiveness of SCA tools (Plösch et al., 2009; Heckman and Williams, 2008; Nanda et al., 2010; Shen et al., 2011). The higher the precision and recall are, the more effective the analysis is. Also the term *accuracy* is sometimes used (Heckman and Williams, 2008, 2011). These measures are defined as the following:

$$precision = \frac{true\,positives}{all\,warnings} = \frac{TP}{TP + FP} \tag{2.1}$$

$$recall = \frac{true\,positives}{all\,known\,bugs} = \frac{TP}{TP + FN} \tag{2.2}$$

$$accuracy = \frac{correct\,classifications}{all\,classifications} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

All the measures rely on the correct classification of warnings as false positives. However, classifying a warning as a false positive is actually a quite difficult task. Nanda et al. (2010) mention in their study that even though an alert is a true positive indicating a real defect, users might ignore the warning for several different reasons. Developers may, for example, think that the defect is unlikely to cause a failure in practice or the defect is reported in the code that is not so critical. More reasons for ignoring these warnings are described by Ayewah (2010), including reasons such as dead code, deliberate errors, and infeasible statements. In fact, using the term false positive

for these legitimate but uninteresting warnings is inaccurate. Therefore, some studies (Heckman and Williams, 2011; Ruthruff et al., 2008; Liang et al., 2010) prefer to use terms *actionable alert* and *unactionable alert*. False positives are naturally unactionable alerts, however, also true positives which developers see not worth fixing are classified as unactionable. Only warnings which developers are willing to fix are considered actionable alerts.

2.2.2 Actionable alert identification techniques

As we have seen in the previous subsection, static analysis tools produce a large number of false positives, or more precisely, only a part of the alerts are actionable. Therefore, several different actionable alert identification techniques (AAIT) have been proposed in the literature. Most of these techniques either re-prioritize warnings so that the most important warnings are at the top or directly classify an alert as actionable or unactionable without prioritization. Heckman and Williams (2011) present a very comprehensive study about AAITs. In this section, we will briefly summarize the general AAIT approaches presented in the aforementioned paper. The included eight approaches are: 1) alert type selection, 2) contextual information, 3) data fusion, 4) graph theory, 5) machine learning, 6) mathematical and statistical models, 7) dynamic detection, and 8) model checking.

Alert type selection

Alert type selection is probably the most straightforward and easiest approach to use. It basically means excluding bug patterns which developers do not see important. Usually, static analysis tools group warnings to categories which users can exclude from analysis. Although the method is simple and might significantly reduce the number of unactionable alerts, it is prone to suppress also those important warnings which belong to the excluded categories. It is very project-specific which types of warnings are relevant. Therefore, this approach requires studying the alert types appearing in the target code base usually by mining the source code repository or manually reviewing different alert types.

Contextual information

AAITs utilizing the contextual information approach try to diminish unactionable alerts by selecting only areas of code that a static analysis tool can analyze well. For example, if we know that some part of a program contains old legacy code which works well but produces many unactionable alerts, we can exclude this legacy portion of the code from the analysis.

Data fusion

Data fusion uses outputs from multiple static analysis tools and combines them to produce more accurate results. Similar warnings from multiple tools might imply that an alert is actionable. For example, the output from FindBugs could be enhanced with the output from PMD. If both FindBugs and PMD report a warning for the same line of code, this warning might get a much higher priority than if only one tool reports a warning for the line of code.

The AAIT presented by Meng et al. (2008) uses the data fusion approach. The authors combined the results from FindBugs, PMD, and JLint. They first prioritized alerts by the priority from each tool. Next, they analyzed whether multiple alerts from different tools point to the same issue. If more than one tool has found the same issue, the alert's rank is raised. The authors did not, however, do any precise evaluation about the effectiveness of this approach.

Graph theory

AAITs using the graph theory approach try to identify actionable alerts by using, for instance, system dependence graphs, which provide control and data flow information for a program. Based on this information, one can calculate, for example, the execution likelihood for some location of code. If a warning points to the line of code having low execution likelihood, we can either ignore the alert or give it a lower priority.

The AAIT described by Boogerd and Moonen (2008) uses the graph theory approach by analyzing program's execution likelihood combined with execution frequency. The higher execution likelihood and execution frequency the code has, the higher are warnings from SCA tools

prioritized.

Machine learning

One of the most common approaches used in AAITs is machine learning. Heckman and Williams (2011) define it as *"the extraction of implicit, previously unknown, and potentially useful information about data"*. In practice, AAITs using machine learning commonly use the information about the alerts and the surrounding code. These AAITs try to find patterns within gathered data and then build a constantly updated model from the data, which can be used to predict or re-prioritize alerts. Usually these AAITs apply some well-known machine learners, such as Bayesian network or logistic regression.

For example, Heckman and Williams (2009) propose a model building process which gathers alert characteristics (AC) from three sources: a static analysis tool, a metrics tool, and a source code repository. The model includes candidate ACs, such as cyclomatic complexity of the method containing an alert, alert priority, alert open revision, total alerts for revision, and file age. From the set of candidate ACs, only those ACs that have the best predictive power are included and irrelevant characteristics are ignored. Using machine learning algorithms and the selected set of predictive ACs, a model is build, which can then be used to predict whether an alert is actionable or not. Heckman and Williams evaluated this AAIT with two subject programs and concluded that the average precision of the AAIT is 89–90% and the average recall is 83–99%. In other words, 89–90% of the alerts classified with the model as actionable were actionable alerts. Furthermore, 83–99% out of all known actual actionable alerts were classified as actionable.

Mathematical and statistical models

Using mathematical and statistical models is another very common approach among AAITs. This approach usually uses the data from alerts and the history from a source code repository to create mathematical or statistical models in order to determine whether an alert is actionable or not.

For example, one AAIT (Kim and Ernst, 2007b) using mathematical and statistical models analyzes alert lifetimes. The basic principle in

the aforementioned study is that alerts fixed quickly are more important to developers. The authors believe that based on this principle and the historical data from the source code repository, they can predict actionable alerts with the aid of mathematical and statistical models. Another AAIT using mathematical and statistical models is the method described by Ruthruff et al. (2008). Their approach to predict actionable alerts is based on logistic regression model which uses 33 ACs identified by the authors. This AAIT was evaluated on 1652 alerts from Google's bug database, where approximately 56% of the alerts were actionable. The authors concluded that the model was able to predict actionable alerts over 70% of the time.

Dynamic detection

Dynamic detection is a less commonly used approach among AAITs. It combines static analysis with dynamic analysis. For example, results from static analysis can be used to automatically generate dynamic test cases. Furthermore, these test cases can be used to verify the faulty behavior of the location of the code identified by an alert from static analysis tool.

Model checking

Another less frequently used approach among AAITs is model checking. This approach combines static analysis and model checking. Model checking can be, for instance, used to better identify concurrency problems. An AAIT using model checking might, for example, create a model from the program under analysis and prioritize the states in the model by analyzing the program code. If a static analysis tool reports a concurrency issue in one of the high-priority states, the alert is more likely to be actionable. One major problem with this approach is that, in large programs, the state explosion problem might make model checking unusable.

2.3 Evaluating the effectiveness of FindBugs

Several different approaches have been used to evaluate the effectiveness of FindBugs in the literature. The most common methods appear

to be evaluating how well FindBugs could have prevented known field defects and measuring the false positive rate of FindBugs. Some studies also have investigated the cost-effectiveness of FindBugs. In the next subsections, we present the previous work about these different approaches.

2.3.1 Detecting field defects

Wagner et al. (2008) examined whether FindBugs could have prevented real documented field defects. They studied two projects: a mature large project having 600,000 LOC and a just finished smaller project having 40,000 LOC. The authors used two approaches to analyze the effectiveness of FindBugs. First, they manually analyzed whether reported bugs from a bug tracker could have been detected with FindBugs. Second, they applied FindBugs for the different revisions of the selected projects and analyzed whether fixed warnings were related to field defects. From the larger project, they could not find a single case where a field defect correlates to a warning generated by FindBugs. For the smaller project, they only used the second approach and found four out of 24 warnings (16.7%) being removed because they actually caused a failure.

More positive results were reported by Nanda et al. (2010). They analyzed the distribution of bugs in two projects: Apache Ant and a commercial project. They estimated that static analysis could catch 5–15% of the reported bugs, most of them being NPEs. One should note that the authors did not verify this estimate by running FindBugs for the selected projects; therefore, the conclusions are questionable.

Kim and Ernst (2007a) studied the precision and recall of FindBugs, JLint, and PMD in three subject programs. They first identified bug-related lines by mining change log messages from the software history having fix change identification keywords (such as "fix" or "bug"). If a warning from a tool matches any bug-related line, the warning is considered as correct. Otherwise, it is a false positive warning. Using this approach, FindBugs got the precision of 5–18% from these three subject programs when including all warning priorities. The recall 2–5% was reported only as the sum of all included tools, however, this means that FindBugs could have prevented 5% of the field defects at best.

To sum up, there seems to be not so many studies about how effectively

FindBugs can detect field defects. Among the few studies, the results vary greatly and are not always comparable. There seems to be at least two practices how to evaluate the effectiveness of FindBugs to detect field defects: comparing warnings from FindBugs to bug reports from an issue tracker and comparing warnings from FindBugs to bugs mined from the commit messages of a version control system.

Wagner et al. (2008) evaluated the effectiveness of FindBugs using the first approach and counted the "*number of warnings related to reported bugs / number of all bug reports in a bug tracker*". However, this measure completely ignores unreported bugs and depends much on the quality of the bug tracking database. Because FindBugs detects many simple defects, which might not get submitted to a bug tracker, this measure may disfavor FindBugs in terms of effectiveness.

We think that the second approach (comparing warnings to bugs mined from a version control system), is a more accurate method because it also includes unreported bugs. This is the method used in the study by Kim and Ernst (2007a). However, instead of manually analyzing the actual cause of a bug, the method used in the aforementioned study relies on automatically marking code lines as bug-related lines based on commit messages from a version control system. Kim and Ernst admit that this approach may produce unrealistic results if developers check in both a fix and many other changes in a single commit.

Categorizing warnings as true positives is another noteworthy issue which varies between the studies. For example, Kim and Ernst (2007a) rely on automatically categorizing a warning as a true positive based on the bug-related lines calculated from the software change history. Wagner et al. (2008) also utilized program change history to count true positives, however, they categorized a warning as a true positive if a warning was removed between selected revisions and it related to a documented failure. Wagner et al. used also another method to calculate true positives: manually examining randomly selected portion of reported bugs and analyzing whether a warning from FindBugs relates to any of the reported bugs.

Clearly, because there are so few studies about the effectiveness of FindBugs to detect field defects, and the results vary much, it is hard to precisely conclude how many field defects FindBugs really can find. If we use the lowest and highest value from the studies, FindBugs is able to detect 0–15% of the field defects.

2.3.2 Measuring the false positive rate

Araújo et al. (2011) studied the effectiveness of both FindBugs and PMD. They first evaluated the tools by analyzing the source code of the Eclipse platform with both FindBugs and PMD. In the study, a warning was considered to be relevant if its lifetime was inferior than the selected time threshold (12, 24, and 36 months thresholds were used). When using the default settings, in the best scenario, only 29.4% of the high-priority warnings of FindBugs were relevant. However, when restricting the analysis to warnings in the correctness category only, the relevance percentage increased to 64%. The authors further extended the study to 12 other open source Java systems, including such popular systems as Struts2, JEdit, Pdfsam, Jython, and Tomcat. In these five systems, FindBugs achieved relevance rates superior to 40%. These are quite good results when compared to the results they got with PMD: the rate of relevant warnings reported by the tool ranged from 2.5% to 10.1% when analyzing the Eclipse platform.

Kim and Ernst (2007a) evaluated the precision and recall of FindBugs, JLint, and PMD in three subject programs. In their study, FindBugs got the precision of 5–18% from these three subject programs when including all warning priorities. In other words, the authors concluded that the false positive rate of FindBugs is as high as 82–95%.

Kester et al. (2010) wanted to know how good is static analysis to find concurrency bugs. The authors used FindBugs for 12 example benchmark programs containing known concurrency issues and concluded that from the total of 13 concurrency bugs, FindBugs can detect four bugs (31%). FindBugs produced total of 12 multithread related warnings and 50% of them pointed to a real bug.

Ayewah and Pugh (2010) conducted a large scale engineering review at Google, involving hundreds of engineers and thousands of warnings from FindBugs. In this study, Google's code repository was analyzed with FindBugs. For two days period, more than 700 engineers ran FindBugs from dozens of offices. They reviewed almost 4,000 warnings from the total of 9,473 warnings. Developers filed more than 1,700 bug reports and submitted code changes that made more than 1,000 of the warnings to disappear. Over 77% of the reviewed warnings were classified as "Must Fix" or "Should Fix". Although the precise accuracy of FindBugs is hard co calculate from these values, it is clear that FindBugs clearly provided relevant warnings which developers were willing

to fix.

Another study from Ayewah and Pugh (Ayewah et al., 2007) analyzed the effectiveness of FindBugs with JDK 1.6. FindBugs reported a total of 379 medium and high priority correctness warnings. The authors reviewed each warning and reported that 56% of the warnings has functional impact. In other words, 56% of the warnings are considered as as actionable alerts. Only 1% was classified as bad analysis from FindBugs. The rest 43% were classified as unactionable alerts not being possible or likely to have little or no functional impact. Based on analyzing the program's build history, the authors also concluded that FindBugs detects issues that developers are willing to address.

We think that using the software change history alone to classify warnings as true and false positives is somewhat questionable. Ayewah et al. (2007) concluded that more than 50% of the warnings removed between the different builds of JDK disappeared because of small targeted changes trying to remedy the issues described by the warnings. Quite the opposite results are described by Wagner et al. (2008). When analyzing two software projects, they concluded that the majority of the warnings removed were due to code changes that were not directly related to the warnings. Moreover, also Kim and Ernst (2007a) concluded that few warnings are removed as the result of actual bug fixes. When analyzing FindBugs, JLint, and PMD in three subject programs, they found out that at best only 9% of the warnings were removed by a fix-change when analyzing 1–4 years of the software change history.

This clearly raises a concern that is it reasonable to assume that warnings which are removed between two software revisions are those warnings which developers are willing to remove—or are these alerts removed just as the side-effect of fixing or refactoring some unrelated other issues. Moreover, when analyzing the effectiveness of FindBugs based only on the software change history, we miss those true positive open alerts which relate to bugs which are not yet fixed.

To sum up, there is clearly much dispersion in the results about measuring the false positive rate of FindBugs: the rate varies between 36–95%. There are several reasons why the results are so different. First of all, some studies only include medium and high priority warnings, while others include also low-priority warnings. Second, there are no clear rules how to categorize a warning as a true positive. Some studies manually analyze the warnings and based on their subjective evaluation categorize an alert as a false or true positive. Other studies

rely on the software change history and make the classification automatically based on whether an alert is removed between two software revisions. Third, some studies limit analyzed warnings to the specific categories of FindBugs, whereas others include all possible categories. For example, when restricting the analysis to warnings in the correctness category only, the true positive rate increased substantially from 29% to 64% in the study by Araújo et al. (2011). Naturally, the analyzed projects also affect the outcome. A more mature project might have completely different false positive rate than some recently finished project.

There is also much dispersion in the terminology used in the studies. For example, among the studies, a true positive warning is defined at least using three alternate terms: a relevant warning, an actionable alert, and a warning having functional impact. Moreover, Ruthruff et al. (2008) defines the term false positive as an alert which is a tool error that does not reveal a real defect and an unactionable alert as an alert which points to a real defect but developers ignore it despite its legitimacy. Instead, Heckman and Williams (2009) use terms false positive and unactionable alert as synonyms for each other. This jumble of terms does not help to compare the results. We think that the terms *actionable alert* and *unactionable alert*, which Heckman and Williams (2011) use in their study, are the most descriptive and should be preferred instead of true positive. Furthermore, the term *false positive* should only refer to cases where an alert is a tool error which does not reveal any defect.

Because there are quite few comparable studies about measuring the false positive rate of FindBugs, it is hard to conclude the precise false positive rate, or more precisely the unactionable alert rate, of FindBugs. Based on the described studies, the rate is somewhere between 36–95%. We believe that better results can be achieved by project-specific customization, for example, by limiting selected bug categories to only those that are important for the project.

2.3.3 Other effectiveness analysis

The effectiveness of FindBugs can also be analyzed based on other measures than detecting field defects or calculating the false positive rate. For example, Wagner et al. (2005) studied the cost-effectiveness of FindBugs. They estimated that detecting a single severe defect or 3–15

normal defects is enough for an SCA tool to be cost-effective. Also Jaspan et al. (2007) concluded that when comparing FindBugs to manual testing in eBay, FindBugs has a better cost-benefit ratio.

In addition to statistical studies described in the previous subsections, the effectiveness of FindBugs is also studied with surveys. Ayewah (2010) reports in his Ph.D. thesis about interviews with experienced developers using FindBugs. Over 1,000 developers responded to the query and 90% of them stated that their investment in FindBugs has been worthwhile. Furthermore, 81% told that FindBugs has found serious problems from their projects.

2.4 Enhancing FindBugs with custom detectors

Not much research has been conducted about enhancing FindBugs with custom project-specific detectors although several studies (Khare et al., 2011; Wagner et al., 2008; Kim et al., 2006; Liang et al., 2010; Nanda et al., 2010) have pointed out that defects occurring in the field are predominantly logical project-specific bugs, which are difficult to find with static analysis tools using default detectors. For example, Khare et al. (2011) analyzed defects present in a navigation system and concluded that 33% of the bugs could have been captured using static code analysis, however, 90% of these defects could not have been detected by off-the-shelf tools because most of the tools do not verify system-specific rules relevant to the domain. We will not go to any technical details how to develop custom detectors for FindBugs, but in this section, we focus on describing the previous results about using the custom detectors of FindBugs. More detailed information how to develop custom detectors for FindBugs is available in the presentation by the authors of FindBugs (Hovemeyer and Pugh, 2007a).

Shen et al. (2008) developed custom FindBugs detectors for projects using AspectJ. This set of detectors, which they call XFindBugs, supports 17 bug patterns to cover common error-prone features in an aspect-oriented systems. The authors tested the detectors with three mature large-scale AspectJ applications: AJHotdraw, AJHSQLDB, and GlassBox. They detected seven reported bugs and found 257 previously unknown defects which may result in a software crash. Clearly, custom detectors are well suited for these types of applications using common

frameworks. Security issues are another area where static analysis suits well because security problems are hard to detect by other means. Ware and Fox (2008) wrote a few security-specific detectors for FindBugs, however, they did not analyze their effectiveness in any way.

Al-Ameen et al. (2011) explored which common bugs FindBugs is not able to find. They noticed that numerous simple bug patterns exist which are not yet detected by FindBugs. The authors developed eight new bug detectors including detectors for such bug patters as zero or negative length array, division by zero, and never executed for loop. They tested the effectiveness of their new bug detectors with five large Java applications (e.g. Android SDK, jEdit, and Spring security) and found over 1,000 issues with the average percentage of false positives being only 15.45%.

To summarize, although there are quite few studies about using custom bug detectors with FindBugs, it seems to be a cost-effective way to enhance the effectiveness of FindBugs. The study by Kim et al. (2006) supports this opinion. The authors of the aforementioned study analyzed bugs in five open source projects and concluded that 19–40% of the bugs appear repeatedly. Because many of the bugs are project-specific, developing custom bug detectors for these frequently occurring defects appears to be reasonable.

It seems that custom detectors suit well for finding violations of project-specific coding rules and misuse of internal or external APIs. Usually, project-specific bug detectors achieve much lower false positive rates than common detectors because project-specific detectors can be better targeted for specific applications and must not be so generic.

Chapter 3

Case study background

In this chapter, we present a brief introduction to the background of the case study performed at Valuatum. We first describe the company background and the background of the system we analyzed in the case study. Second, we provide reasons for why we have selected FindBugs as the static analysis tool for our study.

3.1 Company and system background

Valuatum is a small Finnish software company founded in year 2000. The company delivers equity research solutions and systems to stockbrokers, investment banks, private equity companies, and asset managers worldwide. Customers include such companies as Danske Bank, Evli Bank, Nordnet, and Solidium. Valuatum currently has about 10 employees from which three are software developers. Many different software developers have worked in the company during the 12-year history of Valuatum. Therefore, despite the number of the software developers currently working in the company is small, the code is actually written by many programmers, making it a quite good subject for research. The author of this thesis has worked four years in the company and can be considered as an expert in understanding the system. This actually is another aspect why this system is a good research subject. Previous studies about the effectiveness of FindBugs have usually analyzed systems the authors of the papers have not developed, making it hard for the authors to decide whether a warning from an SCA tool is actionable or not.

The key product of Valuatum is a Java-based enterprise web-application, which the stockbrokers and their clients use, for example, to update the key figures of analyzed companies or to search for the best investment with the aid of the ranking tool. The web-application basically consists of two main modules named *ValuBuild* and *webapp*. The first one contains all the business logic written in Java. The second one contains mostly user interface related code written in JSP, JavaScript, CSS, and XML. To get a general idea about the size of the web-application, alone the ValuBuild module contains about 425,000 NCLOC (non-comment lines of code).

The whole web-application runs on top of the JBoss application server supported by the Struts and Spring frameworks. Although the modern frameworks are in use, there is still much legacy code in the system. For example, some old JSP pages contain both HTML, Java, and SQL in the same file, making them hard to test. Also the ValuBuild module contains old Java code which is not using the aforementioned frameworks but some old hard-to-test custom code. This legacy code is error-prone for changes in the code because no automated tests exist for this old code. Although refactoring the old code and creating automated tests for the new refactored code would be the ideal solution for the problem with the legacy code, the company currently does not have enough resources for that. Therefore, we hope that static analysis could help reducing the bugs in the legacy code by catching at least the most common errors.

The latest changes made to the system are updated to the production environment during the process called *production update*, which is usually done approximately every other month. This process includes heavy testing before updating the code to the production servers. The company uses many well-known development practices in every day development. For example, code reviews, daily scrum, Kanban, unit testing, and automated web tests are all common practices used in Valuatum. Build process is completely automated with the Jenkins continuous integration server.

3.2 Tool selection

In this section, we present the reasons why we have chosen FindBugs as the SCA tool for this case study. The original plan for this study

was to compare the effectiveness of different static code analysis tools in order to select the best SCA tool for Valuatum. However, after performing an initial comparison with the tools, we quickly realized that FindBugs is currently the only eligible free static analysis tool for Java which can find real defects from the program code. Therefore, focusing on evaluating and improving FindBugs was chosen as the focus of this case study.

In the next subsection, we briefly describe the most popular SCA tools for Java which have been presented in the literature. We compared a total of ten SCA tools. We only included tools which support Java programming language and are not only focused on single types of bugs. For example, CheckThread[1], although supporting Java, only finds concurrency issues and is thus not described here in any more detail. Eight of the described tools are free SCA tools and the remaining two are industry-leading commercial SCA tools. We evaluated each tool mainly based on the following viewpoints: project activity, tool focus, SCA methods used, expandability, integration to development environment, and tool license.

3.2.1 Evaluated static code analysis tools for Java

FindBugs

FindBugs[2] is one of the most popular static analysis tools for Java. According to the website of FindBugs, sponsors include large companies, such as Google and Sun Microsystems. FindBugs is even integrated into commercial static analysis tools such as Coverity Static Analysis and HP Fortify Static Code Analyzer (Hovemeyer and Pugh, 2007b).

FindBugs is mainly based on the syntactic pattern matching technique but also includes some data flow features. Most of the bug detectors are based on the visitor design pattern and usually analyze the structure of a class by examining the visited fields and methods. Another common approach among the detectors is to perform linear code scan by analyzing Java bytecode instruction at a time to drive a state machine. More complex detectors use interprocedural data flow analysis to take both control and data flow into account. More detailed description of the tool is available in the Ph.D. thesis by the project founder, David

[1] http://www.checkthread.org/
[2] http://findbugs.sourceforge.net/

Hovemeyer, from the University of Maryland (Hovemeyer, 2005).

FindBugs is more focused on real bugs rather than stylistic issues. It can detect more than 380 bug patterns (Ayewah and Pugh, 2010), including such issues as null pointer dereferences, security flaws, concurrency problems, and performance issues. Since FindBugs operates on the Java bytecode, it can also analyze JSP pages because JSP code can be pre-compiled to Java classes. FindBugs prioritizes warnings based on the coarse three-level (high, medium, low) priority, which is nowadays renamed to confidence level in FindBugs. Recently, a more accurate priority called BugRank was introduced, which assigns alerts a BugRank between 1–20 where a lower value means a more critical bug.

FindBugs is actively developed. The latest version 2.0 came out in December 2011, adding new bug patterns, improved accuracy, and a cloud storage for developers to share bug information. Users can easily extend FindBugs by creating their own custom bug detectors. FindBugs can be easily integrated into various IDEs (e.g. Eclipse), software build systems (e.g. Apache Ant), and continuous integration servers (e.g. Jenkins).

PMD

PMD (unofficially known as Programming Mistake Detector)[3] is another very popular static analysis tool for Java. Unlike FindBugs, PMD operates directly on program's source code, not the bytecode like FindBugs does. To be precise, PMD uses Java and XPath to detect bug patterns from the AST (Abstract Syntax Tree) generated from the source code. PMD uses the syntactic pattern matching technique for detecting bugs. It does not have a data flow component, however, according to the future directions on PMD's website, including data flow analysis is planned.

Many of the defects PMD finds are violations of stylistic and design conventions, in other words, the application could still function properly even if the defects were not corrected. PMD finds such defects as missing JavaDoc comments, unused code, if statements without using braces to surround the code block, and classes containing too many methods. Although PMD is more focused on finding stylistic errors, it

[3] http://pmd.sourceforge.net/

can also be used to find real functional bugs. PMD is highly configurable. It can be easily extended with custom detectors, which PMD calls *rules*. PMD is actively developed, and the latest version 5.0-alpha has been released on January 31, 2012. Like FindBugs, PMD also has a very good plugin support which integrates it to various IDEs and software build systems.

Checkstyle

Checkstyle[4] is an open source static analysis tool focused on finding styling issues. It uses syntactic pattern matching and data flow techniques (Novak et al., 2010). Like PMD, also Checkstyle operates on the AST constructed from the Java source code. It also supports creating own bug detectors, which are called *checks*.

Although the main focus of Checkstyle is checking compliance with coding standards, nowadays more and more checks for other purposes have been added. For example, in addition to checking coding standards, Checkstyle provides checks for finding class design problems and duplicate code. Checkstyle is actively developed: the latest version 5.5 was released in November 2011. It is also integrated with plugins to various IDEs and build tools.

JLint

JLint[5] is similar to FindBugs in that it analyzes Java bytecode and performs both syntactic pattern matching and data flow analysis. JLint is claimed to be extremely fast even with large projects: it requires only one second to check all classes. However, one might question that does it perform good analysis when compared to FindBugs, which takes several minutes to analyze large projects.

JLint can detect three types of bugs: synchronization, inheritance and data flow issues. Synchronization category includes detecting deadlocks and race conditions by building a lock graph from the program and ensuring the non-existence of cycles in the graph. Detectors in the inheritance category find problems with class inheritance, such as components shadowing superclass variable names. Data flow analy-

[4] http://checkstyle.sourceforge.net/
[5] http://jlint.sourceforge.net/

sis is used to detect NPEs by calculating possible range of values for expressions and local variables.

JLint is not currently actively developed. In fact, the project seems to be dead and nobody is developing it. The latest version 3.1.2 is from January 2011, and the previous version 3.1.1 was released in February 2010. According to the tool's change log, no major changes has been made to the tool after the release of the version 3.0 in June 2004.

Lint4j

Lint4j[6] is a static analysis tool which analyzes Java source and bytecode to detect defects by performing data flow, type, and lock graph analysis. Lint4j can detect the following types of defects: performance problems, Java language related problems, architectural problems, code portability issues, serialization problems, synchronization issues, and EJB (Enterprise JavaBeans) specification violations. Although Lint4j can find various types of defects, it has considerably fewer detectors than, for example, FindBugs or PMD.

Lint4j is free, however, it is not open source and is licensed under custom commercial license. The tool is well documented, however, not easily expandable. Lint4j has Eclipse and Maven plugins for easier integration to development workflow. As JLint, also the Lint4j project seems to be dead. According to the user manual, the latest version 0.9.1 was released in May 2006.

ESC/Java2

ESC/Java2[7] is a static analysis tool based on theorem proving. It is described in more detail by Flanagan et al. (2002). The tool performs formal verification of the properties of Java source code. In practice, ESC/Java2 allows a programmer to record design decisions with an annotation language (e.g. specifying that a method parameter can not be null). ESC/Java2 analyzes the annotated program code and warns of any inconsistencies between the design decisions recorded as annotations and the actual program code.

ESC/Java2 can produce some useful output even without any anno-

[6] http://www.jutils.com/

[7] http://kindsoftware.com/products/opensource/ESCJava2/

tations. In this case, it looks for errors such as null pointer dereferences, array out-of-bounds errors, and type cast errors. It can also warn about synchronization errors, such as race conditions and deadlocks, in concurrent programs. As many other free static analysis tools, also ESC/Java2 is a dead project. The web page clearly says that ESC/Java2 is no longer under development. The latest version 2.0.5 is from November 2008. A major problem with this tool is that it can only parse Java 1.4 code; thus, cannot understand Java 1.5 programs. This basically prevents using the tool in any modern program written in Java.

Bandera

Bandera[8] is an open source static analysis tool based on model checking. The tool is described in more detail by Corbett et al. (2000). To use Bandera, programmers annotate their source code with specifications describing what should be checked. Bandera generates a model from the Java source code and, with the help of the model, it can verify the program behavior, such as freedom from deadlocks and the absence of null pointer dereferences.

As many other static analysis tools for Java, also the Bandera project is currently not active. The website states that the project is in hibernation, and there are no plans to continue the project. The latest version 1.0a4 is from May 2006. It is questionable whether Bandera can be used in a real industrial project at all. Rutar et al. (2004) reported that they were not able to run Bandera on any realistic Java program, however, the version they used was a rather old one, 0.3b2.

QJ-Pro

QJ-Pro[9] is an open source static code analysis tool supporting over 200 rules, such as detecting ignored return values and checking code quality based on code metrics. Users can also define their own rules. QJ-Pro analyzes program's source code. The static analysis techniques used by the tool are not described, however, it seems to be mainly using syntactic pattern matching.

[8] http://projects.cis.ksu.edu/gf/project/bandera/
[9] http://qjpro.sourceforge.net/

Unfortunately, also QJ-Pro seems to be a dead project. The latest version 2.2.0 is from May 2005. The tool was compared to both FindBugs and PMD by Wagner et al. (2005). Only 4% of the warnings reported by QJ-Pro were true positives, making the tool the least effective of the compared tools. Similar results are described by Ware and Fox (2008).

Klocwork Truepath

Klocwork Truepath[10] is a commercial static analysis tool supporting C, C++, Java, and C# programming languages. It is the static analysis engine that powers the Klocwork Insight toolset. As PMD and Checkstyle, also Klocwork Truepath performs syntactic pattern matching on the AST constructed from program's source code. Klocwork Truepath also uses interprocedural data flow analysis and symbolic logic to detect more complex bugs. It can find various types of defects, such as concurrency violations, web application vulnerabilities, invalid object references, and violations of coding practices. Users can also create their own custom bug checkers. The tool is described in more detail in the white paper by Fisher (2009).

The Klocwork's tool is actively developed: the latest version (9.5) of Klocwork Insight was released in January 2012. Louridas (2006) has reported that Klocwork K7, the predecessor of Klocwork Insight, costs about $20,000 annually for projects up to a half-million lines of code. The current price of the latest version is unknown because the price of the tool is not specified on the Klocwork's website, and the company did not answer to our queries about their pricing policy.

Coverity Static Analysis

Coverity Static Analysis[11] (formerly known as Coverity Prevent) is another commercial static code analysis tool. Like Klocwork Truepath, the Coverity's tool also supports C, C++, C#, and Java programming languages. It can detect bugs such as concurrency defects, performance degradation problems, null pointer dereferences, and security vulnerabilities. Coverity offers little information about the static analysis techniques they use. The website briefly describes a few proprietary

[10] http://www.klocwork.com/products/insight/klocwork-truepath

[11] http://coverity.com/products/static-analysis.html

static analysis methods used by the tool. These methods include: path simulation, statistical analysis, and boolean satisfiability.

Interestingly, FindBugs is nowadays integrated into Coverity Static Analysis. Coverity has fine-tuned the checkers of FindBugs and has integrated the results into their centralized defect management system, Coverity Integrity Manager. Also Coverity does not list their price publicly. However, Binkley (2007) claims that the previous version, Coverity Prevent, costs $50,000. The price is most likely an annual price even though this is not clearly mentioned in the article.

3.2.2 Reasons for selecting FindBugs

Table 3.1 summarizes the most important features of the SCA tools presented in the previous subsection.

Table 3.1: Summary of the main features of the SCA tools for Java

Tool name	Status	Focus	Main methods	License	Expandable
FindBugs	Active	All bugs	SPM[a], DFA[b]	LGPL	x
PMD	Active	Style	SPM	BSD-style	x
Checkstyle	Active	Style	SPM, DFA	GPL	x
JLint	Dead	All bugs	SPM, DFA	GPL	
Lint4j	Dead	All bugs	DFA, lock graph	Free closed source	
ESC/Java2	Dead	All bugs	Theorem proving	Free open source	x
Bandera	Dead	All bugs	Model checking	GPL	
QJ-Pro	Dead	All bugs	SPM	GPL	x
Klocwork Truepath	Active	All bugs	SPM, DFA	Commercial, $20k/year	x
Coverity Static Analysis	Active	All bugs	Proprietary methods	Commercial, $50k/year	x

[a] SPM = Syntactic pattern matching
[b] DFA = Data flow analysis

Among the free tools, the most active and popular tools are FindBugs, PMD, and Checkstyle. Interestingly, they all use rather simple static analysis methods, such as syntactic pattern matching. Although, for

instance, ESC/Java2 and Bandera both use novel and quite powerful static analysis methods, the tools have not gained any widespread usage. Simplicity seems to overrun novelty in this case. One reason for the poor success of novel free static analysis tools might be that the best techniques may have been commercialized for such tools as Klocwork Truepath or Coverity Static Analysis. However, the few results with commercial tools do not indicate that they use any superior static analysis techniques (Emanuelsson and Nilsson, 2008; Ware and Fox, 2008; Mantere et al., 2009).

From the described tools, the commercial tools were the first ones we had to discard because neither Klocwork nor Cloverity answered to our queries about acquiring a free trial of their software for research purpose. Moreover, as pointed out by Hovemeyer and Pugh (2007b), commercial tools are quite problematic in the academic point of view: their proprietary licenses may prohibit from disclosing any information about the capabilities of the tools. This might be the main reason why so few results are available about the effectiveness of these commercial tools.

With the remaining free SCA tools, we performed an initial comparison with Valuatum's system. Free static analysis tools including JLint, Lint4j, Bandera, ESC/Java2, and QJ-Pro did not perform well in our initial analysis. For example, both Bandera and ESC/Java2 did not even work with Valuatum's system because they do not support Java 1.5. Therefore, they can be discarded immediately. Moreover, the results from both JLint and Lint4j were poor in our initial study where we run the tools for the ValuBuild module. For instance, JLint produced 2,414 warnings from which the majority (69%) were quite useless messages like *"Local variable 'x' shadows component of class x"*. Moreover, the remaining warnings seemed not to be as important as what FindBugs (v1.3.8) did found. JLint also does not have any bug categories or priorities, which makes the output of the tool quite difficult to read. Lint4j produced only 252 alerts from which the majority was also seen unimportant. Neither of the tools did not find the critical deadlock bug described in Section 5.1.3 although both tools should find concurrency issues.

Among the free static analysis tools, one feature seems to be particularly common: the lifespan of the tools is short. For example, JLint, Lint4j, ESC/Java2, Bandera, and QJ-Pro are all practically dead projects. Although the tools are yet somewhat usable, investing any man-hours

to configure these tools seems to be questionable if there are no guarantees that the tools will work with future Java versions. Instead, FindBugs, with the lifespan of almost 10 years, is a very active project. PMD and CheckStyle are other actively developed static analysis tools for Java, however, their focus on styling issues is not in our interest if we want to find defects causing real functional errors.

In addition to project activity, also the expandability of the tools had a great impact on choosing the static analysis tool for this case study. Because many bugs in projects are project-specific, we think that the ability to write custom project-specific bug detectors is very important. Among the free tools, FindBugs is one of the easiest tools to expand with custom bug detectors. JLint, Lint4j, and Bandera can not be extended at all, which makes them less tempting.

To conclude, based on our initial comparison with Valuatum's system, tool expandability, tool focus, and project activity, the only viable tool that remains is FindBugs. Comparing the effectiveness of the different static analysis tools in more detail, which was the original plan, seemed quite pointless because we already knew that FindBugs is most likely the tool which can find the largest number of real defects. Therefore, evaluating and improving FindBugs was chosen as the focus of this case study.

Chapter 4

Case study methodology

In this chapter, we introduce the methodology of the case study performed at Valuatum. In other words, we describe how we planned and executed the case study. As we have described in Section 1.4, the focus of this study is to find answers to the following research questions:

1. **RQ1**: How effective is FindBugs in preventing bugs in mature software systems?
2. **RQ2**: What techniques are applicable in mature software systems to find the most important alerts from the large number of alerts reported by FindBugs?

The first research question is answered by analyzing the effectiveness of FindBugs. We use several different approaches for this analysis:

1. **Analyzing how well FindBugs can detect reported defects.** This approach consists of searching reported defects from an issue tracker and inspecting how many of the reported defects could have been detected with FindBugs.
2. **Analyzing how well FindBugs can detect unreported defects.** This approach consists of searching unreported defects from the SVN commit logs and inspecting how many of the unreported defects could have been detected with FindBugs.
3. **Analyzing how well FindBugs can detect missed defects using custom detectors.** This approach consists of identifying those reported and unreported defects which should have been

caught with SCA but were missed by FindBugs. After the identification, we develop custom detectors for the most important missed defects.

4. **Analyzing what types of alerts are removed during the project history.** This approach consists of fetching several different revisions of the ValuBuild module and performing FindBugs' analysis to these revisions. The results are then compared and FindBugs can identify what alerts are removed during the project history.

5. **Analyzing how well FindBugs can find new defects.** This approach consists of performing FindBugs' analysis to the latest version of Valuatum's product and evaluating how many of the alerts are actionable and how many are unactionable or false positives.

The second research question is answered based on experimenting different approaches to deal with the large number of unactionable alerts. These approaches include: defect differencing, integration to IDE, and the use of simple AAITs. All these approaches are described in more detail in the following sections.

4.1 Study design: FindBugs effectiveness

4.1.1 Detecting reported fixed defects

Our first approach to analyze the effectiveness of FindBugs is to study how well FindBugs could have detected known *reported* fixed bugs. This type of approach was also used by Wagner et al. (2008). We started by first selecting all bug reports from our project management tool having the state "Completed". The bug reports were selected from the previous two and a half year time period (from the beginning of June 2009 to the end of October 2011). We limited the time period to the previous 30 months because older bug reports were only available in our obsolete project management system, and searching bug reports from it would have been difficult. We also believe that bug reports from two and a half years is a sufficient time period for this study.

We identified a total of 158 fixed bug reports from our project management tool. This includes bugs from both ValuBuild and webapp

modules because the bugs are not separated between the two modules. From the identified bugs, 26 had to be ignored because they were duplicates or incorrectly categorized as a bug fix (some of the bugs were clearly more feature improvements than bug fixes).

When removing the inadequate bug reports described above, the final set of bug reports we used was 132 bug reports. Each of these bugs were classified to the following categories:

- **System-specific logical bugs** (e.g. logical errors in algorithms, calling wrong API methods in wrong places, incorrect checks for permissions, wrong data used in wrong places)
- **User interface bugs** (e.g. JavaScript errors, layout issues)
- **Server configuration bugs** (e.g. issues with character encodings, problems caused by server software updates, incorrect library configurations, problems with file system permissions)
- **Common coding errors** (e.g. ArrayIndexOutOfBounds exceptions, unused method parameters, SQL syntax errors, illegal references from JSP pages to Java classes)
- **Third-party bugs** (e.g. JVM errors, issues with external data provider data, problems with external libraries)
- **NullPointerExceptions (NPEs)**
- **Concurrency bugs** (e.g. deadlocks, race conditions)

In addition to assigning each bug to the categories above, we also analyzed each bug report with the following question in mind: Could FindBugs have been used to prevent this bug? We know that, with the default settings, FindBugs can not really detect system-specific logical errors. Neither can it detect user interface bugs, such as layout issues or JavaScript errors. However, FindBugs should be good at detecting, for example, common coding errors, NPEs, and concurrency issues.

We examined the detailed description and comments from each bug report in order to determine the cause of the bug. If the reason for the reported bug was clearly a logical system-specific error, we did not even try to detect it with FindBugs because we know that FindBugs does not understand the internal logic of the system. Instead, if the bug was caused by some coding error other than clear system-specific bug, we

marked it as "SCA feasible", indicating that static code analysis could be a feasible method for detecting the bug.

For each of these "SCA feasible" bugs, we retrieved the revision of the class which contained the bug from SVN, built the project, and checked whether FindBugs can detect the bug by running the FindBugs analyzer for the whole system. The GUI of FindBugs was then used to search for warnings pointing out the bug.

We further analyzed the bugs which we marked as "SCA feasible" and roughly assigned each bug an impact factor ranging from low (minor bug) to very high (critical bug). The impact factors are described in more detail below:

- **Low**: Fixing the bug can wait longer. Users can use the system but with minor disturbance (e.g. a small cosmetic layout issue or some minor administrator functionality not working as expected).
- **Medium**: The bug needs to be fixed in the next production update (a few weeks). The bug causes some deviation from intended behavior and is visible to regular users but no immediate action is needed to fix the bug.
- **High**: The bug should to be fixed within a few days. Users can still use the system but some important part of the system is not working as expected.
- **Very high**: The bug must be fixed immediately. Users can not use the system at all or some critical part of it. The bug might cause data corruption, the return of incorrect data or considerable performance issues.

Section 5.1.1 presents the results about how well FindBugs could have detected known reported fixed bugs when using the approach described above.

4.1.2 Detecting unreported fixed defects

Our second approach to analyze the effectiveness of FindBugs is to study how well FindBugs could have detected known *unreported* fixed bugs. In the previous section, we described an approach for analyzing the effectiveness of FindBugs by mining *reported* bugs from our

project management tool. However, we believe that some of the defects FindBugs can detect do not get reported because bugs found with SCA are usually quite simple. Thus, a developer might just fix the bug and ignores submitting a bug fix task to the project management tool. Therefore, we searched bugs also from our source code management tool (SVN).

To identify the bug fixes, we searched SVN commit messages containing the following words (case-insensitive): Fix, Bug, NullPointer, Null Pointer, and NP. These messages were searched from the trunk and the tags branches from the previous two years time period (from 2009-10-01 to 2011-10-01). The tags branch was included besides the trunk because the tags branch mainly contains bug fixes detected after the production testing phase; thus, it is a good source for discovering bug fixes. We only searched fixed defects from the ValuBuild module. The webapp module was not included because it contains mainly UI code, and FindBugs is known to detect poorly user interface errors.

After fetching all relevant bug fixes from the SVN commit messages, we applied exactly the same steps as with the reported bugs in Section 4.1.1. In other words, we first classified the bugs to categories based on their bug type. Second, we identified all bugs being feasible to detect with SCA and assigned those bugs an impact factor ranging from low (minor bug) to very high (critical bug). Last, we studied whether FindBugs could have detected those bugs which were marked as "SCA feasible".

One should notice that some of the bug fixes identified using the above-mentioned approach might already be in the set of reported bug fixes from our project management tool described in Section 4.1.1. Identifying the duplicates between the reported and the non-reported bugs would be laborious and inaccurate because not all SVN commit messages contain the task id used in the project management tool. Therefore, it would be difficult to determine which commit relates to a reported bug fix. However, this does not really matter because one of our interests is to know the ratio of bugs which can be detected with static code analysis. Including the duplicates does not change this ratio significantly. Furthermore, we are interested in what types of bugs does and does not FindBugs detect. Again, duplicates do not matter here.

This type of approach was also used by Kim and Ernst (2007a), however, the authors of the aforementioned study relied on automatically categorizing a warning as a true positive based on the bug-related lines

calculated from the software change history. Instead, because the author of this thesis knows the system under analysis very well, we rely on manually analyzing each bug fix commit and manually evaluating whether FindBugs could have prevented the bug.

Section 5.1.2 presents the results about how well FindBugs could have detected known unreported fixed bugs when using the approach described in this subsection.

4.1.3 Detecting defects using custom detectors

The results in Sections 5.1.1 and 5.1.2 reveal that FindBugs fails to detect many critical bugs although those missed defects should be quite easy to detect with SCA. Fortunately, FindBugs provides rather easy tools for developing custom detectors. Therefore, we evaluated the effectiveness of FindBugs by developing a few custom detectors to catch the most important missed bugs.

All detectors of FindBugs are written in Java. FindBugs provides an extensive library for creating various types of detectors. The most basic bytecode scanning detectors in FindBugs are based on the visitor pattern. For example, the class `OpcodeStackDetector`, which is the base class for many detectors, has methods `visit(Method)` and `sawOpcode(int)`. While FindBugs analyzes a class, it calls the method `visit(Method)` when the contents of a method are walked. Likewise, FindBugs invokes the `sawOpcode(int)` method as it analyzes each opcode within the method body. When developing a custom detector, one can override these methods and provide a custom implementation.

From the set of bugs FindBugs failed to detect in Sections 5.1.1 and 5.1.2, we selected the most important three defects which we thought would be feasible to catch with FindBugs. We selected the bugs based on the following principles:

1. **The bug is hard to detect by other means.** For example, NPEs are quite easy to detect with good unit testing; therefore, developing a complex detector for the missed NPEs might not be cost-effective. On the contrary, concurrency and performance issues are extremely difficult to detect with traditional testing; therefore, they are good candidates for custom detectors.

2. **Writing the detector should be cost-effective.** In other words,

writing a custom detector for the bug should take hours or days, not weeks or months. Additionally, this also means that it is not sensible to write a custom detector for one low impact defect which has occurred only once and is highly unlikely to occur again. Conversely, good reasons exist for developing a custom detector for a single high impact defect because the cost of fixing the bug might be considerably high.

3. **The detector can be made generic.** Although targeted project-specific detectors might be effective, even better would be if the detector is generic so that it could also be used in other projects.

Following the principles defined above, we developed custom detectors for the following bugs: *Deadlock with static synchronized methods*, *Stateful singletons*, and *Incorrect lazy initialization of class fields*. Each of the bugs and the developed detectors are described in more detail in Section 5.1.3

4.1.4 Detecting defects based on alert removal history

In the previous subsections, we analyzed the effectiveness of FindBugs by searching fixed defects from both the project management tool and the SVN commit messages. Because bugs found by SCA tools are usually quite simple, a bug might get documented in neither the project management tool nor the SVN commit log. Therefore, we applied a fourth approach to study how well FindBugs could have prevented fixed defects: the different versions approach used by, for example, Wagner et al. (2008).

In practice, we analyzed the different revisions of the ValuBuild module and compared the differences between the warnings reported by FindBugs. If a warning has disappeared between two revisions, we know that there might have been a bug which a developer has fixed. Class removals are ignored, so alerts only disappear due to targeted bug fixes or code refactoring. We are especially interested in knowing whether these alert removals are actually targeted bug removals or are they just the side-effect of unrelated code refactoring. This approach might also offer information about the types of warnings developers are willing to fix. In other words, what types of warnings are actionable.

This different versions approach we applied, consists of the following steps:

1. We fetched every 30th revision from the trunk branch starting from revision 18382 (2010-06-14) and ending to revision 20271 (2011-09-13). This totals to 64 different revisions from the last 15 months.
2. We built the project for each fetched revision and analyzed the fetched revisions with FindBugs. If the build did not succeed, we ignored the revision and moved to the next 30th revision.
3. When all revisions were processed with FindBugs, we combined the outputs of FindBugs with the *ComputeBugHistoryTask* of FindBugs. In addition to combining the alerts from multiple software revisions, the task also computes the revision numbers for the alerts when they were first and last seen.

After completing the above steps, we had alert history data from 60 revisions. We had to discard four revisions (18742, 18832, 19162, and 19942) because the build failed in these revisions; thus, FindBugs could not perform the analysis. The starting revision was chosen as 18382 since we could not anymore compile older revisions because of major changes made in the system. Building any older revision would have required extensive manual refactoring to the code.

We identified the removed warnings from the alert history data and categorized these warnings based on their lifetime, BugRank, and bug category. We also manually examined the removed alerts having BugRank 1–14 and the alerts having the lowest lifetime to see whether the removed alerts disappeared because of actual targeted bug fixes—or were they removed because of the side-effect of code refactoring. We classified an alert as an actual targeted bug fix based on the SVN diff between the revisions and also based on the SVN commit message related to the alert removal revision.

The results using the above-mentioned approach are presented in Section 5.1.4.

4.1.5 Detecting open defects

In the previous sections, we described approaches to study the effectiveness of FindBugs to prevent fixed defects. However, despite the many quality assurance practices used, there are still most likely defects in Valuatum's system which are not yet identified. Although these defects probably are not so critical (because nobody has fixed them to date), they might cause considerable problems in certain situations. Especially security problems, concurrency bugs, and performance issues are those types of defects which might hide in the system. Therefore, we also analyze the effectiveness of FindBugs based on examining the open warnings the tool produces.

We executed FindBugs for the latest revision (as of 2012-02-01) of Valuatum's system. Both ValuBuild (business logic in Java) and webapp (user interface code in JSP) modules were included. We did not include our custom detectors described in Section 5.1.3. We used the ant task of FindBugs with the following differences between the default settings:

1. *effort="max"* (enables analyses which increase precision and find more bugs, but may require more memory and take more time to complete)
2. *reportLevel="low"* (includes all possible warnings, not just high and medium level warnings)

Using the GUI of FindBugs and the new cloud plugin, we manually examined each of the scariest warnings and categorized them to the following four categories: *must fix*, *should fix*, *mostly harmless*, and *not a bug*. These are the same categories used by Ayewah and Pugh (2010) in their study at Google. This categorization is also nowadays part of the default user interface of FindBugs.

We used the following guidelines when deciding to which category an alert belongs:

- **Must fix**: A bug described in the alert definitely has some impact on system functionality (e.g. some common functionality is unusable, data gets corrupted or users get unauthorized access to data).
- **Should fix**: A bug described in the alert probably has some impact on system functionality. Less critical and less likely than the

must fix category, however, still a noteworthy defect which should be fixed.

- **Mostly harmless**: A bug described in the alert is highly unlikely to cause any impact on system functionality. In other words, it is highly unlikely that the system ever gets into a state where the bug occurs. And even if that happens, the impact is low. For example, it makes no difference whether the system safely fails for an NPE because a user enters bogus data as URL parameters or, instead, doing null checks and displaying another error message.
- **Not a bug**: A bug described in the alert does not have any functional impact to the system in any scenario. This is definitely a false positive that will never be fixed. Examples in this category are: a null dereference error in case where there is no NPE or a security error in case where there is no security problem in the code. Also unimportant code style issues and code design issues fall into this category.

The results using the above-mentioned approach are presented in Section 5.1.5.

4.2 Study design: Dealing with unactionable alerts

As described in Section 2.2.1, the large number of alerts, and especially the large number of unactionable alerts, is one of the biggest problems when adopting SCA. To overcome this problem, we studied several possible approaches how to deal with the large number of unactionable alerts.

The initial plan was to compare several actionable alert identification techniques (AAITs) described in Section 2.2.2. We admit that some of the AAITs might be quite effective in distinguishing unactionable alerts from actionable. However, after investigating the different AAITs, we realized that many of the approaches might be too complex for Valuatum and might not even produce accurate results. The most effective AAITs usually use mathematical models to predict actionable alerts. Not only does the initial construction of the model take time, but it also takes time to maintain, evaluate, and tweak the model.

Moreover, the results in Section 5.1.4 imply that, in mature software systems, one can not use the history of alert removals as the basis for an accurate prediction whether a future alert is actionable or not. So, to train effective models, we would have to manually review and categorize a large number of alerts.

Therefore, we experimented other more simple approaches to deal with the large number of unactionable alerts in mature software systems which have not used SCA tools before. The following approaches are included:

- **Defect differencing.** This approach consists of displaying developers only alerts which are new when compared to the last known working published program revision. We think that preventing new bugs for occurring is more important than fixing those alerts which has been in the system for a long time. Therefore, we decided to ignore the "old" alerts in every day development and focus on new alerts, which are the ones that might point to new bugs. Section 5.2.1 presents in more detail our experiences in applying this technique to our development process.
- **Integration to IDE.** This approach consists of integrating FindBugs to the developers' desktop environment in order to get rapid feedback. IDE integration should help to detect defects earlier and it also helps to deal with the large number of alerts. We tested IDE plugins for both Eclipse and NetBeans. We have reported our experiences in more detail in Section 5.2.2.
- **Simple AAITs.** This approach consists of using the most simple actionable alert identification techniques described in Section 2.2.2 to deal with the high number of unactionable alerts. We used such AAITs like contextual information and alert type selection, which enabled us to reduce the number of unactionable alerts significantly. The results are described in more detail in Section 5.2.3.

4.3 Software used

In this case study, the version of FindBugs we have used is FindBugs 2.0. This was the latest stable version of FindBugs at the time when

this case study was started. The analysis runs were always performed from the ant task of FindBugs. Also the GUI of FindBugs was used when inspecting the alerts. We have always used the default settings of FindBugs if not mentioned otherwise.

Various custom-made XSLT 1.0 documents were used to filter the bug fix commits from the SVN history dumps and to transform the SVN history dumps from XML to a more human-readable HTML table format. In addition, a few custom-made shell scripts were used to fetch, build, and analyze the different revisions of the ValuBuild module in the different versions approach described in Section 4.1.4.

4.4 Limitations

We have identified several limitations which might have an effect on the results obtained in this case study. In this section, we discuss about the most important limitations we have recognized.

First of all, we have only analyzed one mature software system in this case study. Therefore, from the results, it is hard to draw any strong conclusions which apply also in other mature systems. Nevertheless, we think that Valuatum's system is a quite good subject for study because of the long history and the large number of developers who have worked on the system.

Also the number of known fixed bugs we have analyzed in Sections 5.1.1 and 5.1.2 is quite small. The reason for this is that we had to limit the time period we used to approximately two years. Using a different time period might produce different results. More software projects and more bugs should be analyzed in order to draw any stronger conclusions from the results.

Another limitation is the quality of the issue tracking system we have used when evaluating the effectiveness of FindBugs to detect reported defects. Bug reports are not always so descriptive that we can be sure what the root cause of the bug is. Therefore, we might have missed some bugs which might be feasible to detect with SCA. Moreover, although all bugs in Valuatum's system should be recorded to the issue tracking system, developers most likely have not documented all defects to the issue tracking system. We have tried to overcome this limitation by including also unreported defects from the SVN commit logs. However, also the quality of the commit logs can affect the results:

commit messages are not always so descriptive and a single commit might have contained many bug fixes or other code refactoring. These might have affected our analysis to identify the root cause of each bug; therefore, we might have missed some defects detectable by SCA.

One should note that not all reported and unreported defects analyzed in Sections 5.1.1 and 5.1.2 are defects occurred in the field—some might be defects which have been detected before production update, for example, with the aid of manual testing or automated unit testing. Identifying those defects which have occurred in the field would have been too laborious. Moreover, a well-known fact is that detecting a defect as early as possible in the development process saves money. Therefore, we think it is also important to include those defects which have not occurred in the field but are caught, for example, by manual testing before production update. We might be able to catch these defects even earlier.

In Section 5.1.4, we have analyzed the removed alerts from the history of the ValuBuild module. One should note that we have only considered the trunk branch, which might affect the results—especially the calculated alert lifetimes. More precise analysis would have required to include also the history of ValuBuild's tags branches because bug fixes are usually committed more quickly to the tags branch than to the trunk branch. However, including the tags branches would have complicated the analysis process; therefore, we only used the trunk branch. Also our sample size, every 30th revision, might affect the results. If an alert is removed very fast (e.g. within 1–2 revisions), our every 30th revisions sample size might be too coarse to record the creation and the removal of the alert. Therefore, we might have missed some short-lived alerts.

In Section 5.1.5, we have categorized open defects as actionable and unactionable alerts. Although we have deep knowledge about the analyzed system, we admit that this is a quite subjective classification. One might get different results if the categorization is performed by some other developer. It is also hard to make a difference between the "Must fix" and "Should fix" categories we used. On the other hand, it does not really matter here because bugs in the both categories are bugs that should be fixed. Alerts in the "Must fix" category should be given a higher priority, though.

Chapter 5

Case study results

5.1 FindBugs effectiveness

5.1.1 Detecting reported fixed defects

Our first approach to analyze the effectiveness of FindBugs is to study how well FindBugs could have detected known reported fixed bugs. We fetched bug reports from our project management system and studied how many of the reported bugs could have been detected with FindBugs. Section 4.1.1 describes the process in more detail.

Table 5.1 shows the distribution of the bug reports from our project management system. The second column presents the absolute number of bugs found for each category and the same value as percentage in parenthesis. In the third column, we can see how many of the bugs in each category could even be detected with static code analysis. The percentage in the third column is calculated from bugs in a category, not from the total bugs. The last column shows how many of the bugs are actually detected with FindBugs.

Table 5.2 shows more detailed information about what types of bugs we classified as "SCA feasible" and which of these bugs were detected with FindBugs. Because of confidential requirements, we can not reveal very detailed bug descriptions but have to stay at rather general level.

Table 5.1: Known reported bugs prevented with FindBugs.

Category	Bugs	SCA feasible	FindBugs
System-specific logical bugs	61 (46%)	1 (1.6%)	0 (0%)
User interface bugs	27 (20%)	0 (0%)	0 (0%)
Server configuration bugs	25 (19%)	0 (0%)	0 (0%)
Common coding errors	7 (5%)	5 (71%)	1 (14%)
Third-party bugs	6 (5%)	0 (0%)	0 (0%)
NPEs	5 (4%)	5 (100%)	1 (20%)
Concurrency bugs	1 (1%)	1 (100%)	0 (0%)
Total	**132 (100%)**	**12 (9.1%)**	**2 (1.5%)**

Table 5.2: Known reported bugs categorized as SCA feasible

Bug type	Impact	FindBugs	Notes
Common coding errors			
Unused parameter in constructor	Very high	Yes	FindBugs reports a warning "Unwritten field" for this bug giving BugRank 15.
ArrayIndexOutOfBounds exception	Medium	No	Calls `vector.elementAt()` to an empty vector.
BigInteger misusage	Low	No	`BigInteger` class was used to create a hexadecimal string. `BigInteger.toString(16)` omits leading zeros.
Invalid Java class reference in a JSP page x 2	Medium	No	Change in a Java class broke a reference in a JSP page. Can be detected by pre-compiling JSP pages.
Concurrency bugs			
Deadlock with static synchronized methods	Very high	No	Deadlock was caused by using static synchronized methods in two static classes using each others.
Logic			
Stateful singletons	Low	No	Incorrectly using attributes in a singleton class which creates a state for the singleton. FindBugs does not have a detector for singleton classes.
NPEs			
NPE in a JSP page x 3	Medium	No	Requires effective interprocedural analysis.
NPE in a Java class	Very high	Yes	Not detected by default. Detected by adding the CheckForNull annotation which generated many false positives.
NPE in a Java class	High	No	Requires effective context-sensitive analysis.

The results above do not look flattering from FindBugs' point of view. As we see from Table 5.1, from the total 132 bugs, we estimated that 9% could be caught by using static analysis. This would have been a quite good percentage. However, FindBugs only detects two of these bugs, which means that FindBugs detects only 1.5% of the selected 132 bugs. To be precise, without using annotations, FindBugs can only find one of the total 132 bugs, and this one warning also gets a quite low priority from FindBugs (BugRank 15). Because of the low priority, developers probably would have ignored this alert. On the other hand, the impact of this single bug was very high, so even preventing this one single bug might have made FindBugs a cost-effective tool.

Our results are quite similar to the results from other studies. First of all, we estimated that 9% of the bugs could have been detected with SCA. This is very close to the results from Nanda et al. (2010) who evaluated two software projects and concluded that static analysis could catch 5–15% of the reported bugs. The actual percentage of bugs we detected with FindBugs is, however, much lower: 1.5%. Most likely, also Nanda et al. would have obtained similar results if they had verified their estimates with FindBugs. This 1.5% of the reported bugs—or 2 out of 132 bug reports—is very close to the results from the study by Wagner et al. (2008). The authors of the aforementioned study concluded that none of the reported field defects could have been detected with FindBugs. Interestingly, the software project analyzed by Wagner et al. was quite close to the product of Valuatum in terms of software size and project maturity.

The results we have obtained reveal that FindBugs fails to detect quite many rather simple bugs which should be detectable with SCA. For example, many NPEs are not detected. Clearly, the NPE detectors of FindBugs should be further improved. Another example of a simple missed defect is calling `vector.elementAt()` for an empty vector, which causes an exception. We think that this could be detected with FindBugs using data flow analysis. Although vector is an obsolete collection, the same detector could be used with `ArrayList` and other collections.

Evidently, it seems that reported defects can not be effectively detected with FindBugs. Based on our results and the results from the previous studies, the percentage of reported bugs detectable by FindBugs is somewhere between 0–2%. We think that there are several reasons why this percentage is so low. First of all, at least in Valuatum, most

of the reported bugs are system-specific logical bugs, UI bugs or server configuration issues, which are all hard to detect with SCA. Second, the issue tracker might not contain bug reports for all simple bugs detectable with SCA, which might distort the results significantly. Therefore, in the next subsection, we analyze how well FindBugs can find unreported defects.

5.1.2 Detecting unreported fixed defects

In the previous section, we analyzed the effectiveness of FindBugs by mining reported bugs from our project management tool. In this section, we present the results of evaluating the effectiveness of FindBugs by searching bugs from SVN change log messages. Using the approach described in Section 4.1.2, we identified a total of 201 fixed bugs (135 from the trunk branch and 66 from the tags branch). Table 5.3 shows the distribution of the unreported fixed bugs. As in the previous section, the table also presents how many of the bugs in each category are even feasible to be detected with static code analysis (3rd column) and how many of these bugs are actually detected with FindBugs (last column).

Table 5.3: Known unreported bugs prevented with FindBugs.

Category	**Bugs**	**SCA feasible**	**FindBugs**
System-specific logical bugs	121 (60.2%)	5 (4.1%)	0 (0.0%)
User interface bugs	35 (17.4%)	0 (0.0%)	0 (0.0%)
Server configuration bugs	11 (5.5%)	0 (0.0%)	0 (0.0%)
Common coding errors	13 (6.5%)	10 (76.9%)	2 (15.4%)
Third-party bugs	3 (1.5%)	0 (0.0%)	0 (0.0%)
NPEs	17 (8.4%)	17 (100%)	10 (58.8%)
Concurrency bugs	1 (0.5%)	1 (100%)	0 (0.0%)
Total	**201 (100%)**	**33 (16.4%)**	**12 (6.0%)**

Table 5.4 shows a more detailed information what types of unreported bugs we classified as "SCA feasible" and which of these bugs were detected with FindBugs.

Table 5.4: Known unreported bugs categorized as SCA feasible

Bug type	Impact	FindBugs	Notes
Common coding errors			
ArrayIndexOutOfBounds	High	No	Same as in Table 5.2
RuntimeException	High	No	Converting byte to Integer.
Self assignment to method parameter	Very high	Yes	FindBugs reports "Dead store to local variable...".
Bad SQL grammar	High	No	FindBugs does not scan for SQL grammar mistakes.
Return value ignored	High	Yes	Detected only with Check-ReturnValue annotation (BugRank 13).
BigInteger misusage	Low	No	Same as in Table 5.2
Empty if clause and missing return statement	Medium	No	If clause was empty instead of containing a return statement
ArrayIndexOutOfBounds	Low	No	Called `String.deleteCharAt` for position -1.
Removing a wrong object from a list	High	No	Mistakenly used an int value instead of an Integer object to remove an object from a list.
Typecast error	Medium	No	Typecasting an object to String.
Concurrency bugs			
Deadlock with static synchronized methods	Very high	No	Same as in Table 5.2
Logic			
Absurd null/empty String comparison	Medium	No	An internal API call should be threated as null/empty check.
Mismatch between Spring XML and class field names	Medium	No	FindBugs does not understand Spring XML
Non-null method expected to return null	Medium	No	FindBugs did not find this even with NonNull annotation.
API misusage	Medium	No	Using a null value as a parameter for non-null method.
Incorrect lazy initialization of class cache	Medium	No	Mistakenly used `!=` instead of ==.
NPEs			
General NPE in a Java class	High	Yes	FindBugs reports "Redundant nullcheck..." with BugRank 20.
NPE in an often used project class x 7	Medium	Yes	When adding the CheckForNull annotation FindBugs reports "Possible null pointer..." (BugRank 13)
NPE in logging mechanism	High	Yes	Detected only with the CheckForNull annotation but generated a huge number of false positives
NPE when passing null to non-null parameter	Medium	Yes	FindBugs reports an alert with BugRank 6
General NPE in a Java class x 7	Medium	No	Requires context-sensitive and interprocedural analysis.

Clearly, from the FindBugs' point of view, there is some improvement when compared to the results in the previous subsection. First of all, the number of identified bugs increased from 132 to 201 although we used a half-month shorter time-period. Second, the percentage of bugs categorized as SCA feasible increased from 9.1% to 16.4%. Third, the number of defects detected by FindBugs increased from 1.5% to 6%. So, at the first glance, it seems that analyzing the effectiveness of FindBugs to detect defects produces too poor results if we include only reported bugs. However, one should note that most of the unreported bugs FindBugs is able to detect are detected with the aid of annotations. If not using annotations, the total number of detected bugs decreases from 12 to 3. This means that only 1.5% of the total number of identified bugs can be detected with FindBugs without using annotations. Although 1.5% is not a flattering percentage, this is still more than in the previous subsection. One should note that the priority levels of the warnings are quite low; therefore, developers probably would have ignored most of the alerts. Only one alert exists which is in the scary category (BugRank 1–9).

The results described in Table 5.3 are similar to the previous studies described in Sections 2.3.1 and 2.3.2. Our estimate that 16.4% of the bugs could be detected with SCA is quite close to the 5–15% which Nanda et al. (2010) estimated when analyzing the distribution of bugs in two software projects. Moreover, we experimented that FindBugs can detect 6% of the unreported bugs, which is very close to the results from Kim and Ernst (2007a) who concluded in their study that FindBugs could have prevented maximum of 5% of the field defects identified from the version control system.

When comparing the bug categories in Tables 5.1 and 5.3, there seems to be clearly more system-specific logical bugs within unreported bugs (60%) than reported bugs (46%). Interestingly, some of the system-specific logical bugs could be feasible to detect with SCA. Not a great surprise is that the number of server configuration bugs has decreased quite a lot, likely because these types of bugs are usually fixed directly at the servers; thus, no trace is left to the version control system.

Perhaps the most interesting thing is that the number of NPEs has considerably grown. Most likely, this increase is due to the fact that small NPE fixes are not documented to the issue tracker but directly fixed to the code and uploaded to the version control system. Because SCA is good at finding NPEs, this increase in NPEs explains much why

the percentage of SCA feasible bugs has increased when compared to the results with the reported fixed defects. Clearly, the NPE detection of FindBugs could still be improved. The tool misses quite simple NPEs and sometimes does not find simple NPEs even with the help of annotations.

To sum up, based on the previous studies and the results from the case study presented in this subsection, it seems that FindBugs can detect 1–6% percent of the unreported bugs. The results depend much on whether you use annotations or not. Especially NPE detection can be improved with annotations. Several reasons exist why this percentage remains rather low. First of all, about 83% of the identified bugs are system-specific logical bugs, UI bugs or configuration issues, which are hard to detect with SCA. Second, we think that to get the most out of SCA tools, the tools should be integrated into IDEs because there might be common bugs which never get from developers' desktop environment to the version control system. We also think that the effectiveness of FindBugs can be further improved by developing custom detectors, which we discuss in more detail in the next subsection.

5.1.3 Detecting defects using custom detectors

As we have seen in Sections 5.1.1 and 5.1.2, FindBugs failed to detect many critical bugs, although those missed defects should be quite easy to detect with SCA. In this section, we describe the results from developing custom detectors for three of the missed defects. Reasons why we have chosen these three bugs are described in more detail in Section 4.1.3. These custom detectors can be downloaded as a single package from the following URL: `http://iki.fi/mvestola/thesis/detectors.jar`.

Deadlock with static synchronized methods

The bug described in this section was the most critical and laborious bug found from Valuatum's system during the recent years. As calendar time, it took one full month to detect the root cause of this bug. We had three experienced developers working on this bug and we have estimated that it took about 100 man-hours altogether to find and fix this bug. Developing the detector for this bug took about 8 hours. Thus, the time-savings would have been massive if we had used this detec-

tor in our testing process. The bug also caused noticeable harm for our customers because it randomly caused the servers to stop responding. All in all, this bug caused serious trouble for both developers and customers—not to mention the negative effects it caused to the company's image.

The root cause of this bug was that the system contained static Java classes with static synchronized methods calling one another. Because static synchronized methods use the class itself as a lock object, this causes deadlocks in some situations. Consider the example illustrated in Figure 5.1. In the example, classes `A` and `B` both contain static synchronized methods named `method1()` and `method2()`. Furthermore, `A.method1()` uses `B.method1()` and `B.method2()` uses `A.method2()`

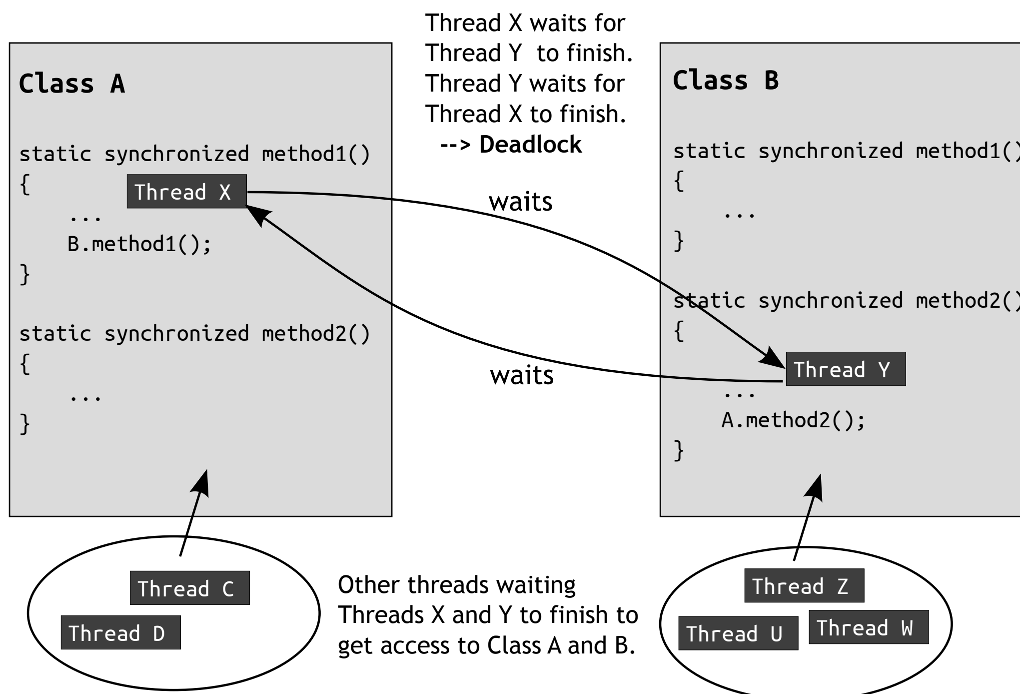

Figure 5.1: An illustration of the deadlock with static synchronized methods

If thread `X` is at method `A.method1()` and thread `Y` is at `B.method2()`, thread `X` needs to wait access to `B.method1()` because thread `Y` is already in `B.method2()`, holding the lock for all static synchronized methods in class `B`. Moreover, thread `Y` also needs to wait access to `A.method2()` because thread `X` holds the lock for all static synchronized methods in class `A`. Therefore, both threads wait each others, causing a deadlock.

This does not only cause threads `X` and `Y` to be in the deadlock state but also other threads trying to access static synchronized methods in either class `A` or `B` will end up waiting.

Because of the nature of concurrency bugs, it was extremely difficult to find the root cause of this bug. The only trace for this bug was that because of the many threads waiting in the deadlock state, the number of active database connections multiplied and finally reached the maximum allowed value thus preventing the server to connect to the database. However, this sudden increase in database connections was first seen as the cause for the servers to crash, not the symptom of something else. The bug was finally traced to the deadlocks by analyzing the thread dumps of the servers.

The custom detector we made to detect this bug is rather simple. It uses the two-pass technique of FindBugs. In the first pass, the prerun detector scans every static synchronized method call found from the program under analysis and constructs a shared database containing information about from which methods static synchronized method calls are invoked. In the second pass, the post-run detector again scans every static synchronized method call found from the program and—using the shared database constructed in the previous pass—determines whether deadlock is possible when calling the method.

Deadlock is possible if all of the following is true (considering the previous example):

1. The current parent method `A.method1()` is static synchronized
2. The current invoked method `B.method1()` is static synchronized
3. The class of the invoked method (`B`) contains a static synchronized method which invokes any of the static synchronized methods in the class `A`.

There are some known limitations for this detector. One limitation is that the detector can not identify deadlocks in complex chains of method calls. For example, if classes `A`, `B`, and `C` contain static synchronized methods, the following chain of calls can cause a deadlock which the detector does not currently detect: `A.method1()` calls `B.method1()`, which invokes `C.method1()`, which invokes `A.method2()`. Although the detector is not perfect, it definitely is able to detect the deadlock bug described in Figure 5.1 and does not produce any false positives. The

detector did not find any new deadlock bugs from the most recent code of Valuatum. Last but not least, this detector is not system-specific but is generic; therefore, it can be used in any system to improve deadlock detection.

Stateful singletons

Unlike the deadlock bug described in the previous section, the bug described in this section is not so critical. However, this bug was chosen because detecting this concurrency bug is difficult by other means. Another reason was that the bug has occurred in several classes. Moreover, not only one developer has introduced this bug but the bug seems to be quite common among all developers. Although several classes contain this bug, it has not yet caused any critical problems because the bugs appear mainly in features which are currently not so heavily used.

The root cause of this bug is that the system contains Java classes which are defined as singleton objects. In other words, if a class is a singleton, only one instance of the class exists in the whole system (within the same class loader). In Valuatum's system, singletons are defined as singleton beans in Spring XML files. A class defined as a singleton bean can not have fields creating a state for the object. If a class contains fields which can create a state, this may cause unexpected concurrency problems if two threads use the singleton bean at the same time. The worst case scenario is that a user may receive corrupted response or get access to unauthorized data.

Like the deadlock detector described in the previous section, also the detector for this singleton bug is quite simple. It first creates a database of singleton classes based on Spring XML files. Second, the detector scans every class which is defined as singleton for fields which may change the state of the class instance. For example, if the singleton class has a non-final private field, this field definitely creates a state for the class instance and is considered to be a bug. However, final immutable fields (such as primitive values and instances of the String class) and fields which are defined as singletons are allowed because they can not change the state of the class instance.

It took only about 8 hours to write this detector which is a cheap price for detecting this type of hard to find concurrency issue. The detector found a total of 26 real bugs from 11 different classes when we applied

it to Valuatum's system. Many of the bugs were previously unknown. Although the testing for the detector was not so extensive, it seems to work well and does not produce any false positives. The detector is not as generic as the deadlock detector described in the previous section. However, it can be used as such in other systems using the popular Spring framework.

Incorrect lazy initialization of class fields

The detector described in this subsection finds incorrect use of lazy initialization in the getter methods of classes. For example, the following code example is considered to be a bug because the comparison "`field != null`" does not make sense here. Instead, it should be "`field == null`".

```
 private SomeObject getField() {
    if (field != null) {
       field = new SomeObject();
    }  return field;
}
```

This error in lazy initialization might cause an NPE which is, however, usually easy to detect with traditional testing. Nevertheless, the bug might additionally cause performance problems if the field is accidentally initialized elsewhere and initializing the object requires heavy computation. This type of performance problem is much more difficult to detect.

Developing the detector to find this bug took about 8 hours. The detector basically scans the program code for a specific sequence of Java bytecode. It also uses the field and the method name to determine whether the analyzed method is a getter method for some field of the class. The detector seems to find at least the most basic bugs and does not to produce any false positives, however, the testing was not so exhaustive.

5.1.4 Detecting defects based on alert removal history

When using the approach described in Section 4.1.4, FindBugs identified that a total of 752 warnings were removed between the selected starting and ending revision. When ignoring warnings being removed by class removals (using the setting "`removedByChange = true`" in FindBugs), the total number of warnings decreases to 692.

Table 5.5 reveals the distribution of the warnings based on the lifetime of the alerts. The lifetime here means how many analyzed revisions the warning has lasted before it was removed.

Table 5.5: Distribution of removed warnings based on warning lifetime

Lifetime	Warnings (count)	Warnings (%)
0	87	12.6%
1–2	47	6.8%
3–9	97	14.0%
10–15	114	16.5%
16–29	65	9.4%
30–49	104	15.0%
50–60	178	25.7%
Total	**692**	**100.0%**

Because we only included every 30th revision in our analysis, one step in lifetime actually means a step of 30 revisions. If the lifetime is 0, it means that the warning was first introduced at revision x and last seen at the same revision x. In other words, the warning was not preset at the next analyzed revision $x + 30$. Therefore, we know that a warning having lifetime 0 was fixed in 30 revisions or less. Furthermore, if the lifetime is 3, the warning was first introduced at revision y and last seen at revision $y + 30 * 3$. This means that a warning having lifetime 3 was removed between revisions $y + 90$ and $y + 120$ thus having the actual lifetime between 90 and 120 revisions. The main point with the lifetime is that the lower value means that the warning is fixed faster.

We further classified each warning to Table 5.6 based on the BugRank, which is the new priority ranking of FindBugs. It is basically an enhancement to the older three-level priority classification. The lower

the BugRank is, the more confident we should be that the warning is a real defect. The table below also contains the average lifetime for each BugRank scale.

Finally, we also classified the warnings to Table 5.7 based on the warning categories from FindBugs. The table below also contains the average BugRank and the average lifetime for all categories.

Table 5.6: Removed warnings classified based on BugRank

BugRank	Warnings (count)	Warnings (%)	Avg. lifetime
Scariest (1–4)	0	0%	-
Scary (5–9)	12	1.7%	16.3
Troubling (10–14)	23	3.3%	31.2
Concern (15–20)	657	94.9%	24.7
Total	**692**	**100.0%**	**24.8**

Table 5.7: Removed warnings classified based on warning category

Category	Warnings	Warnings	Avg. BugRank	Avg. lifetime
Bad practice	100	14.5%	18.7	15.9
Correctness	24	3.5%	9.5	20.3
Experimental	6	0.9%	20	34.2
I18N	7	1.0%	19.6	28.6
Malicious code	34	4.9%	17.8	15.9
MT correctness	3	0.4%	13.0	35.3
Performance	329	47.5%	18.2	31.5
Security	18	2.6%	13.4	45.8
Style	171	24.7%	19.4	16.4
Total	**692**	**100.0%**	**18.1**	**24.8**

At the first glance, it might seem that FindBugs could have prevented 692 defects, which is the total number of warnings removed during the selected 15 months time period. This would be a really good result for FindBugs. However, a closer look reveals that most of the removed warnings are rather unimportant.

First of all, more than half of the warnings has quite a long lifetime. More precisely, as we can see from Table 5.5, over 50% of the warnings has lifetime longer than 15 analyzed revisions. If we make a theoretical assumption that revisions are equally distributed to the 15 months (450 days) time period we have used, it means that, on an average,

developers have created 4.2 new revisions per day. Therefore, the 15 analyzed revisions lifetime means that an alert has lasted at least 450 revisions, which equals to three and a half months. To continue, more than a quarter of the alerts has lifetime 50 or more, which equals at least 1500 revisions or a full year. These are rather long lifetimes for alerts indicating bugs.

The results described in Table 5.7 also imply that most of the removed alerts are quite unimportant. Almost $3/4$ of the removed alerts belong to the performance or style category, both categories having a very low average BugRank. Style issues rarely indicate actual bugs. In the performance category, 81% of the warnings were alerts about creating `Integer`, `Double` or `String` objects inefficiently (e.g. `new Integer(int)` instead of `Integer.valueOf(int)`). This certainly impacts performance to some extent, however, we think that it is still rather unimportant.

As we can see from Table 5.6, none of the fixed warnings belong to the scariest category (BugRank 1–4). Most of the warnings (94.9%) have the lowest BugRank 15–20. Moreover, these warnings with the lowest BugRank were on an average more quickly fixed than warnings having the troubling BugRank. There might be several reasons why the analyzed project history did not contain a single removed alert from the scariest category. First, FindBugs might not have many detectors reporting high BugRank alerts. Second, the high BugRank alerts might have been removed so fast (e.g. within 1–2 revisions) that our every 30th revisions sample size is too coarse. Third, high BugRank alerts might indicate so critical bugs that they are removed by using other QA practices before the code is committed to the SVN.

We manually analyzed each removed warning having BugRank 1–14. After inspecting the total of 35 removed alerts, we identified that only 6 (or 17%) of these removed alerts were actual targeted fixes to fix the problem FindBugs has detected. The rest of the alerts were removed because of extensive method refactoring where an alert was removed just as a side-effect for code refactoring. Interestingly, not all of the targeted bug removals were removed fast. We identified one removed alert having lifetime 4, one having lifetime 3, and one having lifetime 1. Rest three bug removals had lifetime 0. The bug fixing process used in Valuatum might explain the rather long alert lifetimes. In Valuatum, a field defect is usually fixed first to the tags branch, which contains small patches to be updated to the production environment. These patches are not necessarily immediately merged to the trunk

branch but after a few weeks or so. Because we have only analyzed the alert removals from the trunk branch, this might explain the long lifetimes for some important alert removals.

We also manually inspected each of the removed alerts having lifetime 0. From the total of 87 removed alerts, only 6 (or 6.9%) were targeted fixes to fix the problem detected by FindBugs. The rest of the removed alerts were again removed as a result of major refactoring of the code. Most of the removed alerts were rather unimportant alerts, such as invoking `new Integer()` instead of `Integer.valueof()`.

Our results support the results from the existing studies by Wagner et al. (2008) and Kim and Ernst (2007a). Both studies concluded that the majority of the warnings removed from the software projects they analyzed were due to code changes that were not directly related to the warning. Interestingly, Ayewah et al. (2007) reports opposite results by claiming that more than 50% of the warnings removed between different builds of JDK were due to small targeted changes to remedy the issue described by the warning. Ayewah et al. analyzed only alerts from the correctness category and alerts having high or medium priority, which might explain the differences between the results to some extend. However, we also studied the high-priority alerts and most of the analyzed alerts were in fact alerts from the correctness category, and still we got quite the opposite results.

To summarize, it seems that most of the alerts removed during the project history are quite unimportant: the alerts have rather long lifetime and the priority of the alerts is quite low. Moreover, most of the alerts are not removed as a result of actual targeted bug fixes but because of unrelated code refactoring. This implies that, at least with mature software systems—such as Valuatum's system—one can not use the history of alert removals as the basis for an accurate prediction whether a future alert is actionable or not. Some AAITs (Kim and Ernst, 2007b; Heckman and Williams, 2009) described in Section 2.2.2, use this approach, which is quite questionable at least with mature software systems.

5.1.5 Detecting open defects

In this section, we describe the results from analyzing the effectiveness of FindBugs to detect open defects. The approach we have used is described in more detail in Section 4.1.5. When using this approach,

FindBugs produced a total of 12,052 open warnings for Valuatum's system (5,382 for the ValuBuild module and 6,670 for the webapp module). Most of the warnings (94%) were quite low level warnings having BugRank between 15–20. When including only the scary bugs (BugRank 1–9), the number of warnings decreases to 312.

Obviously, over 12,000 warnings is far too much for us to manually analyze. Therefore, in this study, we only included the 312 scariest alerts for more detailed analysis. We manually analyzed each of the 312 alerts and categorized them to the importance categories described in Section 4.1.5. Table 5.8 presents the results in more detail.

Table 5.8: Open alerts having BugRank 1–9 classified based on alert category and importance level

Category	Alerts	%	MF[a]	SF[b]	HL[c]	NB[d]
Correctness	108	34,6%	12	22	63	11
Bad practice	2	0,6%	1	1	0	0
MT correctness	20	6,4%	0	18	2	0
Security	182	58,3%	4	0	177	1
Total	**312**	**100.0%**	**17**	**41**	**242**	**12**
			5,4%	13,1%	77,6%	3,8%

[a] MF = Must fix
[b] SF = Should fix
[c] HL = Harmless
[d] NB = Not a bug

Clearly, when including alerts having BugRank 1–9, FindBugs produces far too many unactionable alerts. Only 18.5% of the scary alerts were classified as actionable (Must or Should fix). The percentage would have been most likely even lower if we had considered alerts having lower BugRank.

Almost half of the warnings are about security issues from JSP pages, mostly XSS issues. However, we think that FindBugs provides too high priorities for the XSS issues in Valuatum's system because most of the issues are non-persistent XSS vulnerabilities in admin pages. Moreover, almost all pages in the system are private password protected. Therefore, an attacker must have a username and a password for the system in order to search for XSS vulnerabilities. Alerts about multi-thread correctness issues are those warnings which are most often

categorized as actionable—possibly because of the fact that the consequences of a multi-thread bug are quite high because they can affect the whole system, for example, by causing deadlocks or data corruption.

One should note that although the total number of unactionable alerts is quite high, FindBugs did find many actual new unknown bugs from the system. All the 17 alerts in the "Must fix" category point to actual bugs which should be fixed as soon as possible because they clearly have functional impact to the system. The "Must fix" category contains such bugs as: ignored method return values, infinite recursive loops, comparing strings using == instead of `String.equals(String)`, NPEs, and XSS vulnerabilities in public pages. If we use the criterion from Wagner et al. (2005), who reported that detecting a single severe defect or 3–15 normal defects is enough for an SCA tool to be cost-effective, we can conclude that these 17 detected defects are enough for FindBugs to be a cost-effective tool.

We also superficially analyzed the warnings from the troubling (10–14) and concern (15–20) ranks. Because of the large number of warnings, the analysis was very superficial; thus, no precise classification is provided. Most of the warnings seemed to fall to the mostly harmless category. However, some of the warnings from the troubling category, such as alerts about SQL injections, raised some concerns. Although most of the alerts about SQL injections seemed to be quite harmless, there were some alerts which definitely need further studying whether an SQL injection is possible. FindBugs also revealed some real performance bugs from the concern category, mainly unclosed database connections, from which one actually caused real problems in the production environment. However, the alert was missed because the BugRank of the alert was so low.

Our results show that only 18.5% of the high-priority alerts are actionable alerts. This is quite close to the results from Kim and Ernst (2007a) who evaluated the precision of FindBugs in three software systems. The authors concluded that only 5–18% of the alerts indicated a real bug. In other words, only 5–18% of the alerts were actionable. Araújo et al. (2011) reported higher precision: 29% of the high-priority warnings of FindBugs were considered actionable. Two studies (Ayewah and Pugh, 2010; Ayewah et al., 2007) reported completely different results. The first study reports experiences from Google. In that study, the authors concluded that 77% of the reviewed warnings were

classified as "Must Fix" or "Should Fix". However, only 42% of the total 9,473 warnings were reviewed so the actual percentage of actionable alerts per all alerts is most likely much lower. The second study reports that as much as 56% of the analyzed warnings from JDK 1.6 had functional impact. The differences between the results can be explained to some extent by the fact that the definition of actionable alert varies in the studies. Some studies classify an alert as actionable automatically based on the project history and alert lifetime, while other studies use manual evaluation, which is prone to subjectivity.

To sum up, many of the FindBugs' alerts about new defects seem to be quite unimportant. Although some studies have reported that more than 50% of the alerts are actionable, based on our results and other previous studies, we believe that most of the alerts are actually unactionable. The actual percentage of actionable alerts is closer to 15–30% when including only high-priority alerts. The percentage is most likely much lower if also low-priority alerts are included. Nevertheless, FindBugs seems to find quite a few new previously unknown bugs which we might have missed otherwise. We see detecting security, performance, and multi-thread issues especially useful because they are hard to detect by other means. We think that to get the most out of FindBugs, we should tweak the FindBugs' settings to match the needs of our project, for example, by excluding unimportant bug categories and classes. We can also use various other approaches to deal with the high number of unactionable alerts, which we discuss in more detail in the next section.

5.2 Enhancing FindBugs: Dealing with unactionable alerts

As described in Section 2.2.1, the large number of unactionable alerts is the biggest problem with static code analysis. Even though a static analysis tool could find real defects, warnings about those defects might get cloaked by unimportant alerts. This is especially problematic when applying SCA tools for mature software systems which have not used SCA tools before. For example, as we have seen in Section 5.1.5, in Valuatum's system, FindBugs produced over 12,000 alerts when first applying the tool, and only 18% of the high-priority alerts were categorized as actionable.

When initiating a new software project, it is quite easy to start using an SCA tool because one can immediately begin following the practices recommended by the tool. However, when introducing an SCA tool for a mature project, the tool usually produces so many alerts that fixing the issues would take too much resources. For example, if one developer is working 8 hours per day, 40 hours per week, and could fix one alert in five minutes, fixing 12,000 alerts would take 25 weeks, which equals to almost half a year.

In this section, we introduce a few methods to deal with the high number of unactionable alerts and describe our experiences with the deployment of these techniques in Valuatum. These presented methods are: defect differencing, integration to IDE, and the use of simple AAITs. The reasons why we have chosen these techniques are describe in more detail in the case study methodology in Section 4.2.

5.2.1 Defect differencing

As the results in Section 5.1.5 imply, most of the alerts from FindBugs are quite unimportant. We believe that although there are some alerts pointing to real bugs, most of the alerts actually do not indicate a bug that has any functional impact to Valuatum's system. If the latest release of the system has been running several weeks without problems, it most likely does not contain any serious bugs. Therefore, we are not so interested in fixing the existing alerts which has first appeared before the latest release revision.

We think that the most important thing is to prevent new bugs for occurring. Therefore, in Valuatum, we decided to ignore the 12,000 "old" alerts in our every day development and focus on the alerts introduced after the latest release revision because these alerts are the ones that might point to new bugs. In this way, we can ensure that at least all added new code and refactored code follow the recommended practices of FindBugs. We certainly will fix the most important alerts we have already identified from the 12,000 "old" alerts—and we might some day go back to reviewing the remaining alerts, however, we do not use these in our every day development process. We believe that it is psychologically better to show only small portion of the alerts than viewing all the 12,000 alerts, which might depress the developers and make them ignoring the tool output. Using the defect differencing approach, we also hope that developers react faster to the alerts.

We have now integrated FindBugs to our continuous integration (CI) process using this defect differencing approach. Below is described in more detail how our process works:

1. Our CI server Jenkins starts a static analysis task automatically every night.

2. The task fetches and builds the latest code from the trunk branch and the latest release branch.

3. The task runs FindBugs for both the ValuBuild module and the webapp module and searches every possible alert from both the trunk branch and the latest release branch.

4. The task calculates alert history using the *computeBugHistory* ant task of FindBugs. This calculation produces one large XML file containing all alerts from the both included branches with the information about when an alert was first introduced. This XML file is filtered with the *filterBugs* ant task of FindBugs to produce the final output file containing only alerts which are new when compared to the alerts from the latest release revision.

5. If new alerts are detected, the task automatically fails the build and sends email to the development team. Users can see the alerts from Jenkins and browse to the line of source code which the alert complains about.

6. The developer whose changes caused the alerts to appear should either fix the code so that the alert disappears or if the developer thinks that there is no defect in the code, he or she should suppress the warning with the annotations of FindBugs. All alert suppressions must be well reasoned to the annotations.

When analyzing the alert history data from Section 5.1.4, we calculated that, during the last year, we released the latest code to the production environment approximately in 150 revisions interval. Between the releases, 80 new alerts were introduced on the average. We manually analyzed some of the new alerts and we think that we can safely decrease the number of new alerts by 50% by filtering out alert types we are not interested in. Thus, we will most likely have to inspect about 40 new alerts between future releases, which sounds reasonable and is

not a burden to the developers because the alerts are fixed every little while during the development process, not all in one. The number of alerts to be inspected will probably even decrease over time when developers start to adapt to the recommended practices of FindBugs.

To get more immediate feedback, it would be better to run the SCA task in Jenkins after each code change, however, the process is quite heavy and slow (takes about 40 minutes). Therefore, we decided to only run it at midnight if we can not find ways to speed up the process.

Not much previous work exists about the experiences in applying this defect differencing approach. Only Nanda et al. (2010) have reported their experiences about using defect differencing in IBM and they think it is a very important feature of their static analysis toolset. To get more information about the effectiveness of using this defect differencing approach, we would need track the usage of the approach for longer period of time. However, at the moment, our development team has a very positive feeling about the approach and the team is committed to use FindBugs.

5.2.2 Integration to IDE

In Section 5.1.2, we studied how many fixed bugs mined from the SVN commit messages could have been detected with FindBugs. We concluded that only few of those bug fixes could have been prevented with FindBugs. This result might indicate that developers find the most important bugs already before they commit their changes to the version control system. This might be the reason why we could not find so many fixed bugs from the SVN which FindBugs could have detected. For example, clear NPEs which cause the system to crash are most likely caught at least during the testing phase, which a developer should always make before committing any changes to the SVN. FindBugs might, however, detect these defects already before the testing phase.

According to Boehm (1981), fixing a defect during the development phase is approximately 10 times cheaper than fixing the defect during the testing phase. Also Wagner et al. (2005) has pointed out that it seems to be beneficial to run SCA tools as early as possible before any code reviews because the automation makes it cheaper and more thorough than using manual reviews. Therefore, we prefer to catch the defects early before any testing or code reviews with an automated SCA tool. The defect differencing approach presented in the previous sub-

section does not address this issue because it runs FindBugs after the code is already tested, reviewed, and committed to the SVN. For that reason, we have now planned to integrate FindBugs to the developers' desktop environment using plugins for IDEs.

This IDE integration approach not only does help to detect defects earlier but also helps to deal with the large number of alerts. Because the IDE plugins of FindBugs show the alerts directly in the code editor, developers can focus on alerts related to only their modified classes and can fix the most common bugs identified by FindBugs before any testing or code review. There exists FindBugs plugins for both Eclipse and NetBeans, which are two of the most popular IDEs.

There are, however, some limitations with the IDE plugins. We first tried IDE integration with NetBeans, which is the IDE we are usually using in Valuatum. We installed the Software Quality Environment (SQE)[1] toolkit for NetBeans 7.1.1. The SQE is not maintained by the developers of FindBugs but it is the only plugin for NetBeans which can run FindBugs. After a few hours of testing, we quickly realized that this plugin is quite unusable. First of all, the plugin always runs FindBugs full analysis when the IDE is opened. This initial run usually takes about 5–10 minutes to finish, takes very much CPU, and at the end, completely freezes the IDE for a few minutes. Second, one can not use the same alert type filters defined for the ant task; thus, providing a shared alert type configuration for the developers is difficult. Third, the plugin does not work with the custom detectors described in Section 5.1.3: the custom detectors are detected after some customization but alerts do not appear in the editor. Last, the plugin uses the old version of FindBugs (1.3.8) instead of the newest 2.0.

Nevertheless, we see that the NetBeans plugin could be very useful if the shortcomings described above are fixed. After the initial run, the plugin detects new problems from the code in a few seconds and displays the alerts to the developer immediately. However, now the plugin is quite unusable in every day development and we probably will not even introduce the NetBeans plugin to our developers since they most likely will disable the plugin because of its annoying shortcomings.

Instead, the FindBugs' Eclipse plugin, which is maintained by the developers of FindBugs, is almost perfect. It does not have the same shortcomings than with NetBeans. In Eclipse, users can easily use custom detectors and reuse filter files from FindBugs' ant task. Users

[1]http://kenai.com/projects/sqe/

can also filter shown alerts based on the BugRank, and the Eclipse plugin is also integrated to the FindBugs' cloud plugin so that users can add comments about the alerts. It takes about 5–10 minutes to run FindBugs for the whole project but users can also easily run FindBugs manually for only a single file or a single package. Furthermore, users can also configure the plugin to be run automatically, which significantly speeds up the process and enables giving rapid feedback to the developers.

To sum up, we think that IDE integration is very important when applying SCA tools. It provides developers rapid feedback and the issues are most likely fixed faster than when inspecting the alerts after the nightly SCA task has been run in the CI server. In order to know if IDE integration really can help to detect more bugs earlier, we should somehow gather usage data from the developers' desktop environments for longer time and evaluate how many and what types of bugs have the IDE plugins reported. We think that both running FindBugs in a centralized CI server and running FindBugs in developers' desktop environment are important. The centralized CI server provides defect differencing and makes sure that important new alerts are not missed. Because the IDE integration of FindBugs with NetBeans seems to not work well, we are currently planning to move on to Eclipse, which has good support for FindBugs.

5.2.3 Simple AAITs

In this subsection, we describe how we used the most simple AAITs described in Section 2.2.2 to deal with the high number of unactionable alerts. The initial number of alerts we started with was the same as in Section 5.1.5, in other words, slightly over 12,000. By using simple AAITs, such as contextual information and category selection, we managed to reduce the number of alerts from over 12,000 to 4,400, in other words, at least 63% of the alerts were seen unactionable and were removed. We admit that 4,400 alerts is still much, however, because we mainly use the defect differencing and IDE integration approaches, this does not matter so much.

We mainly ignored alerts based on the following principles:

- If an alert points to old code which we know is working well even though it contains defects pointed out by FindBugs, we ignore the

bug pattern in the old code.

- If an alert is clearly a stylistic error not relevant in Valuatum (e.g. confusing method names), we completely ignore the bug pattern in all code.
- If an alert is irrelevant because of some framework we use, we ignore the bug pattern in classes using the framework. For example, FindBugs warns about uninitialized fields in some classes, however, these are not relevant alerts because Spring handles the field initializations for these classes.

We discovered that filtering out alerts based on the BugRank of FindBugs is not very effective. Many alerts which we are interested in have very low BugRank. For instance, unclosed database connections have sometimes caused significant problems in Valuatum but BugRanks for detectors detecting these issues are low. Jaspan et al. (2007) reports similar experiences.

Interestingly, some specific bug patterns produce significant number of alerts. For example, the bug pattern which warns about classes using instance variables while extending the `Servlet` class, is completely irrelevant in Valuatum's system—but still FindBugs generated about 2,000 alerts about this bug pattern. This is about 17% of the total number of alerts. Jaspan et al. (2007) has reported similar problems. The authors mention that, in eBay, two FindBugs checkers produced over half of the original issues and all of these alerts were unimportant in the eBay's environment. This might explain the great variance between the results in Section 2.3.2 where we presented previous work about evaluating the false positive rate of FindBugs. We think one should always manually review the warnings at least superficially to see whether there are any bug patterns which certainly produce too many alerts. Otherwise these bug patterns may have too big effect on the results.

To sum up, we think it is very important to configure FindBugs to match the system under analysis. You should always exclude bug patterns which are not meaningful to your team. By using this project-specific tweaking, one can easily achieve more than 50% reduction in unactionable alert rate.

Chapter 6

Conclusions

In this thesis, we have evaluated and enhanced the effectiveness of FindBugs in preventing bugs in mature software systems. We have also explored several applicable approaches to find the most important alerts from the large number of warnings reported by FindBugs.

First, to analyze the effectiveness of FindBugs, we studied how many fixed defects FindBugs could have prevented. Based on the findings from our case study and the previous work by others, we have estimated that FindBugs can detect 0–2% of the reported fixed defects and 1–6% of the unreported fixed bugs. In addition to fixed defects, we also studied the open alerts FindBugs reports for Valuatum's system. We concluded that only 18.5% of the most high-priority alerts are actionable. This is far less than the 50% rates reported by some studies (Ayewah et al., 2007; Ayewah and Pugh, 2010).

Based on the results, FindBugs seems not to be a very effective tool in preventing bugs in mature software systems. We estimated that 9% of the reported fixed bugs and 16% of the unreported fixed bugs could have been prevented with SCA. However, FindBugs does not get even near these numbers. The tool failed to detect many defects which should be feasible to detect with SCA. There seems to be still much room for improvement with FindBugs.

However, judging the effectiveness of FindBugs only based on the percentages above does not do justice for FindBugs. Despite the low percentage of bugs detected, we found FindBugs to be a cost-effective tool because of several reasons. First of all, as presented by Wagner et al. (2008), detecting a single severe defect or 3–15 normal defects is enough

for an SCA tool to be cost-effective. FindBugs was able to detect a few severe fixed bugs and it also detected 17 new, previously unknown important issues which we might have missed otherwise.

We also developed custom detectors for the most important bugs missed by FindBugs. These custom detectors help us preventing known bugs for occurring again. For example, we think that it is highly valuable to have an automatic custom bug detector for the severe deadlock bug described in Section 5.1.3. This ensures that we never have to spend another 100 man-hours to detect and fix this concurrency bug. Based on our experiences, we think that project-specific detectors are very useful and they can be used to significantly improve the effectiveness of FindBugs. Developing custom detectors is especially useful for those bugs which are hard to detect by other means (e.g. concurrency issues).

We have presented several techniques to mitigate the problem with the large number of unactionable alerts. We found out that some of the actionable alert identification techniques (AAITs) presented in the literature use somewhat questionable methods to automatically categorize alerts as actionable based on the alert removal history. Our results imply that, in mature systems, one can not use the history of alert removals as the basis for an accurate prediction whether a future alert is actionable or not. Therefore, some of these AAITs are not really applicable in mature systems. We think that using more lightweight approaches gives good enough results. Defect differencing and IDE integration are examples of methods which we think are applicable techniques in mature software systems to find out the most important alerts. Furthermore, simple AAITs, such as contextual information and alert type selection, are also applicable techniques and can reduce the unactionable alerts easily by 50%.

To sum up, the main goal of this thesis was to learn how to make FindBugs as an effective tool which could provide immediate, useful feedback for developers in Valuatum. We have learned that although FindBugs seems to be ineffective in finding fixed defects, it is able to detect some severe defects, which is enough to make the tool cost-effective. Furthermore, we can improve FindBugs by using custom detectors and by using several techniques to reduce the number of unactionable alerts. We are now using FindBugs in our every day development in Valuatum and the tool has already proved its usefulness by detecting a few of issues which might have caused noticeable problems in our production environment.

Chapter 7

Future work

In this chapter, we present some ideas for future work based on the results from our case study. First of all, it would be interesting to try the AAITs using machine learning methods presented in Section 2.2.2, and to study their effectiveness. For example, evaluating the accuracy of the method described by Heckman and Williams (2009) would be a good topic for further research. We would especially like to know whether there any alert characteristics which could be used in Valuatum to predict whether an alert is actionable or not. Also combining dynamic analysis with static analysis seems to be an interesting approach worth further studying. For example, Chen and MacDonald (2008) present some ideas about using dynamic analysis to improve the effectiveness of static analysis.

Second, we would also like to see more custom project-specific bug detectors. We will most likely continue developing custom bug detectors in Valuatum if we encounter relevant bugs which could be detected with SCA. However, we would like to see custom detectors also from other parties. For example, many commonly used enterprise Java frameworks—such as Spring, Struts or Hibernate—have some specific coding rules that developers should follow. The developers of these frameworks could provide custom FindBugs detectors to the community bundled with their frameworks. In this way, the developers of the community could easily automatically detect violations of framework-specific coding rules.

Third, we would like to see more new static analysis methods implemented into FindBugs. We think that many of the novel static analysis techniques presented in Section 2.1 could be used to improve FindBugs.

However, it is a shame that these new techniques have not gained as wide industrial usage as syntactic pattern matching or data flow analysis techniques. One reason for this might be the weak tool support, which does not encourage developers to use them. We think that researchers should focus on integrating the new techniques to commonly used tools rather than always developing a completely new SCA tool with completely new UI. Developers might be unwilling to integrate a yet another SCA tool with different UI to their development process. However, if the novel techniques were implemented so that they could be integrated into FindBugs, which has good reporting features and IDE integration, maybe these techniques would get more widespread usage and more people get interested in improving them.

Bibliography

M. Al-Ameen, M. Hasan, and A. Hamid. Making FindBugs more powerful. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Service Science*, ICSESS '11, pages 705–708, Beijing, China, July 2011. ISBN 978-1-4244-9699-0. doi: 10.1109/ICSESS.2011.5982427.

J. Araújo, S. Souza, and M. Valente. Study on the relevance of the warnings reported by Java bug-finding tools. *Software, IET*, 5(4): 366–374, August 2011. ISSN 1751-8806. doi: 10.1049/iet-sen.2009.0083.

N. Ayewah. *Static analysis in practice*. PhD thesis, University of Maryland, College Park, MD, USA, 2010. URL http://hdl.handle.net/1903/10934.

N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 1–5, Seattle, WA, USA, July 2008. ISBN 978-1-60558-051-7. doi: 10.1145/1390817.1390819.

N. Ayewah and W. Pugh. The Google FindBugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–251, Trento, Italy, July 2010. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831738.

N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–7, San Diego, CA, USA, June 2007. ISBN 978-1-59593-595-3. doi: 10.1145/1251535.1251536.

N. Ayewah, Y. Yang, and D. Sielaff. Instrumenting static analysis tools on the desktop, February 2010. URL `http://research.microsoft.com/apps/pubs/?id=120327`. Technical Report MSR-TR-2010-17, Microsoft Research.

D. Baca, B. Carlsson, and L. Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 79–88, New York, NY, USA, June 2008. ISBN 978-1-59593-936-4. doi: 10.1145/1375696.1375707.

D. Binkley. Source code analysis: A road map. In *Proceedings of the 2007 Workshop on the Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, May 2007. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.27.

B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981. ISBN 978-0-1382-2122-5.

C. Boogerd and L. Moonen. On the use of data flow analysis in static profiling. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '08, pages 79–88, Beijing, China, September 2008. ISBN 978-0-7695-3353-7. doi: 10.1109/SCAM.2008.18.

G. Brat and A. Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Conference on Aerospace*, pages 1–10, Big Sky, MT, USA, March 2005. ISBN 0-7803-8870-4. doi: 10.1109/AERO.2005.1559604.

J. Chen and S. MacDonald. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '08, pages 8:1–8:9, Seattle, WA, USA, July 2008. ISBN 978-1-60558-052-4. doi: 10.1145/1390841.1390849.

J. Corbett, M. Dwyer, J. Hatcliff, and S. Laubach. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international Conference on Software Engineering*, ICSE '00, pages 439–448, New York, NY, USA, June 2000. ISBN 1-58113-206-9. doi: 10.1145/337180.337234.

P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217(0):5–21, July 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.06.039.

G. Fisher. Klocwork Truepath: Generating accurate, scalable whole program analysis. White paper, Klocwork, July 2009. URL `http://www.klocwork.com/resources/white-paper/tag/truepath/truepath-scalable-whole-program-static-analysis/download`.

C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ISBN 1-58113-463-0. doi: 10.1145/512529.512558.

P. Godefroid, P. de Halleux, A. Nori, S. Rajamani, W. Schulte, N. Tillmann, and M. Levin. Automating software testing using program analysis. *Software, IEEE*, 25(5):30–37, September 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.109.

S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 41–50, New York, NY, USA, October 2008. ISBN 978-1-59593-971-5. doi: 10.1145/1414004.1414013.

S. Heckman and L. Williams. A model building process for identifying actionable static analysis alerts. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 161–170, Washington, DC, USA, April 2009. ISBN 978-0-7695-3601-9. doi: 10.1109/ICST.2009.45.

S. Heckman and L. Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, April 2011. ISSN 0950-5849. doi: 10.1016/j.infsof.2010.12.007.

D. Hovemeyer. *Simple and effective static analysis to find bugs*. PhD thesis, University of Maryland, College Park, MD, USA, 2005. URL `http://hdl.handle.net/1903/2901`.

D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, December 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052895.

D. Hovemeyer and W. Pugh. Using FindBugs for research. Presentation slides of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation. PLDI '07, June 2007a. URL `http://findbugs-tutorials.googlecode.com/files/uffr-talk.pdf`.

D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 9–14, San Diego, CA, USA, June 2007b. ISBN 978-1-59593-595-3. doi: 10.1145/1251535.1251537.

IEEE. IEEE standard 610.12-1990, 1990.

C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 963–970, New York, NY, USA, October 2007. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297964.

D. Kester, M. Mwebesa, and J. Bradbury. How good is static analysis at finding concurrency bugs? In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 115–124, Timisoara, Romania, September 2010. ISBN 978-0-7695-4178-5. doi: 10.1109/SCAM.2010.26.

S. Khare, S. Saraswat, and S. Kumar. Static program analysis of large embedded code base: an experience. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 99–102, New York, NY, USA, February 2011. ISBN 978-1-4503-0559-4. doi: 10.1145/1953355.1953368.

H. Kienle, J. Kraft, and T. Nolte. System-specific static code analyses: a case study in the complex embedded systems domain. *Software Quality Journal*, pages 1–31, April 2011. ISSN 0963-9314. doi: 10.1007/s11219-011-9138-7.

S. Kim and M. Ernst. Which warnings should I fix first? In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The*

Foundations of Software Engineering, ESEC-FSE '07, pages 45–54, Dubrovnik, Croatia, September 2007a. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287633.

S. Kim and M. Ernst. Prioritizing warning categories by analyzing software history. In *Proceedings of the 4th ICSE International Workshop on Mining Software Repositories*, MSR '07, page 27, Minneapolis, MN, USA, May 2007b. ISBN 0-7695-2950-X. doi: 10.1109/MSR.2007.26.

S. Kim, K. Pan, and E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, November 2006. ISBN 1-59593-468-5. doi: 10.1145/1181775.1181781.

G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *Proceedings of the 2010 IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 93–102, New York, NY, USA, September 2010. ISBN 978-1-4503-0116-9. doi: 10.1145/1858996.1859013.

P. Louridas. Static code analysis. *Software, IEEE*, 23(4):58–61, July 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.114.

M. Mantere, I. Uusitalo, and J. Roning. Comparison of static code analysis tools. In *Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE '09, pages 15–22, Glyfada, Greece, June 2009. ISBN 978-0-7695-3668-2. doi: 10.1109/SECURWARE.2009.10.

N. Meng, Q. Wang, Q. Wu, and H. Mei. An approach to merge results of multiple static analysis tools. In *Proceedings of the 8th International Conference on Quality Software*, QSIC '08, pages 169–174, Oxford, England, August 2008. doi: 10.1109/QSIC.2008.30.

M. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2 of *ICSE '10*, pages 99–108, New York, NY, USA, May 2010. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810310.

J. Novak, A. Krajnc, and R. Zontar. Taxonomy of static code analysis tools. In *Proceedings of the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics*, MIPRO '10, pages 418–422, Opatija, Croatia, May 2010.

R. Plösch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer. On the relation between external software quality and static code analysis. In *Proceedings of the 32nd Annual IEEE Software Engineering Workshop*, SEW '08, pages 169–174, Kassandra, Greece, October 2008. ISBN 978-0-7695-3617-0. doi: 10.1109/SEW.2008.17.

R. Plösch, A. Mayr, G. Pomberger, and M. Saft. An approach for a method and a tool supporting the evaluation of the quality of static code analysis tools. In *Proceedings of the 2009 Workshop on Software Quality Modeling and Evaluation*, SQMB '09, Kaiserslautern, Germany, March 2009.

Research Triangle Institute. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, May 2002. ISBN 978-0-7567-2618-8. URL `http://www.nist.gov/director/planning/upload/report02-3.pdf`.

N. Rutar, C. Almazan, and J. Foster. A comparison of bug finding tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Saint-Malo, France, November 2004. ISBN 0-7695-2215-7. doi: 10.1109/ISSRE.2004.1.

J. Ruthruff, J. Penix, J. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 341–350, Leipzig, Germany, May 2008. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368135.

H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao. XFindBugs: eXtended FindBugs for AspectJ. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 70–76, New York, NY, USA, November 2008. ISBN 978-1-60558-382-2. doi: 10.1145/1512475.1512490.

H. Shen, J. Fang, and J. Zhao. EFindBugs: Effective error ranking for FindBugs. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages

299–308, Berlin, Germany, March 2011. ISBN 978-0-7695-4342-0. doi: 10.1109/ICST.2011.51.

S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin / Heidelberg, May 2005. ISBN 978-3-540-26054-7. doi: 10.1007/11430230_4.

S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for Java. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 248–257, Lillehammer, Norway, April 2008. ISBN 978-0-7695-3127-4. doi: 10.1109/ICST.2008.63.

M. Ware and C. Fox. Securing Java code: heuristics and an evaluation of static analysis tools. In *Proceedings of the 2008 Workshop on Static Analysis*, SAW '08, pages 12–21, New York, NY, USA, June 2008. ISBN 978-1-59593-924-1. doi: 10.1145/1394504.1394506.

J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, April 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.38.