An Evaluation of Two Bug Pattern Tools for Java

Stefan Wagner, Florian Deissenboeck
Technische Universität München
Garching b. München, Germany
{wagnerst,deissenb}@in.tum.de

Michael Aichner*
beck et al. projects GmbH
München, Germany
michael.aichner@bea.de

Johann Wimmer
Cirquent GmbH†, München, Germany
{johann.wimmer}@cirquent.de

Markus Schwalb‡
mobileX AG, München, Germany
{Markus.Schwalb}@mobilexag.de

Abstract

Automated static analysis is a promising technique to detect defects in software. However, although considerable effort has been spent for developing sophisticated detection possibilities, the effectiveness and efficiency has not been treated in equal detail. This paper presents the results of two industrial case studies in which two tools based on bug patterns for Java are applied and evaluated. First, the economic implications of the tools are analysed. It is estimated that only 3–4 potential field defects need to be detected for the tools to be cost-efficient. Second, the capabilities of detecting field defects are investigated. No field defects have been found that could have been detected by the tools. Third, the identification of fault-prone classes based on the results of such tools is investigated and found to be possible. Finally, methodological consequences are derived from the results and experiences in order to improve the use of bug pattern tools in practice.

1. Introduction

The automated static analysis of software code has been a subject of research for many years now. The hope is that many quality defects that would need human inspectors or elaborate dynamic tests can be checked automatically. The aim is to reduce the effort needed for finding and removing defects from the software. For various languages there are such tools available now with varying degrees of sophistication. Some tools use advanced techniques such as abstract interpretation while others resort to simple detection of so-called *bug patterns*. These capture common pitfalls in a programming language and thereby help a developer to avoid them. Several tools are able to detect certain bug patterns, e. g. FindBugs [11] or *PC-Lint*[1].

Although significant effort is spent for researching bug patterns and implementing them in tools, there has not been the same amount of research on the effectiveness and efficiency of the tools. Although several efforts were made [15, 18, 19], there are still several open questions. In particular, each of these attempts is only able to analyse a small number of case studies. Especially the wide variety of tools and their differing detection capabilities make an evaluation difficult. Hence, empirical analyses must be repeated and gradually extended to come to conclusive results. Therefore, we concentrated on investigating two bug pattern tools for Java in two industrial projects to improve the empirical knowledge.

Problem The underlying problem is to achieve economically efficient quality assurance for software. Automation is a way for more efficiency and automated analysis using bug pattern tools claims to find defects automatically. Hence, we need to investigate whether such tools can be used to improve the efficiency of defect-detection. In detail, the following questions need to be answered:

1. When is the use of the tools economically reasonable?
2. Can bug pattern tools detect faults detected by other means and field defects?
3. Can fault-prone classes be identified using the tools?

Results The main results are that the analysed bug pattern tools are not well suited to detect potential field defects. The defects occurring in the field are predominantly logical

*Was with TU München at the time of the study
†At the time of the study Cirquent was called Softlab
‡Was with Cirquent at the time of the study

[1]http://www.gimpel.com/html/pcl.htm

faults that cannot be characterised by a bug pattern that is not very domain-specific. The warnings are able to improve the code and support the learning process of the developers but the tools cannot understand the logics of the system. Nevertheless, it is still possible that a small amount of potential field defects are detected and our analysis shows that this can be enough to economically justify the use of such tools. Moreover, we are also able to show that the results of the tools can be used to identify fault-prone classes which in turn can increase the efficiency of other defect-detection techniques.

Contribution The main contribution of this paper is additional empirical knowledge for bug pattern tools in the areas effectiveness, efficiency, and fault-proneness. We are able to quantify the costs of field failures and the use of the tools with real monetary values from practice. Moreover, we give a list of methodological consequences that allow a better use of the tools in practice.

2. Bug pattern tools

The term *bug* has a long history in computer science to denote something erroneous. Today, it is used most commonly as a synonym for *fault* [12]: "An incorrect step, process, or data definition in a computer program. [. . .] In common usage, the terms "error" and "bug" are used to express this meaning."

2.1. Bug pattern

Patterns are a common technique in Software Engineering and other fields to document and reuse specific and important examples. They are most popular in software design [9]. *Bug patterns* use the same idea that specific constructs and constellations in software code lead to defects. Common pitfalls of a programming language are documented so that developers can learn to avoid them. For example, a collection of bug patterns for Java can be found in the book by Allen [1].

A typical example for a bug pattern is the *Split Cleaner* [1] that is used to describe programs that improperly manage resources, either by leaking them or by freeing them too early. This usually is caused by execution paths of the program that free resources like files or database handles not exactly *one* time.

Similar to bug patterns are *anti-patterns* which also denote unwanted situations but relate mainly to architectural patterns. Another related concept is that of "code smells" [8]. These are unclean code, for example very long methods, that is not a fault but contributes to a difficult understanding of the code and hence the introduction of faults. In fact, the distinction between "code smells" and "bug patterns" is sometimes blurry.

2.2. Tool support

Having established bug patterns for a language, the idea to identify them automatically is straightforward. Therefore, there is a variety of tools available that support automatic checking of software code. These tools, that are commonly seen as part of static analysis, are usually able to analyse other issues as well such as conformance to coding styles. The tools contain a set of *detectors* or *rules* that define concretely the bug patterns they identify. These tools do neither distinguish between "bug patterns", "anti-patterns" and "code-smells" nor illustrate which rules actually address defects and which concern maintainability issues.

Commercial software manufacturers as well as the open source community have produced a wide range of bug pattern tools for many popular programming languages. Two well-known examples for Java that are available as open source are *FindBugs* [11] and *PMD*[2]. Popular tools for C, C++ and C# are *PC-Lint*[3], *Klocwork*[4] and *FxCop*[5].

3. Study setup

To answer our research questions, we use three different designs on two tools and two industrial projects.

3.1. Subjects

For the present work we chose to use FindBugs and PMD as these are widely used in the Java world and are open source, which is a requirement in the context of the analysed projects. Additionally, these tools produced the smallest number of false positives in one of our earlier studies [18]. Our quality controlling framework ConQAT[6] [5] is applied to correlate results generated by both. For the analysis we use two different sets of configurations for both tools: (1) the standard configuration (*StdConfig*) and (2) an individual configuration (*IndConfig*), a subset that focuses exclusively on bug patterns and does not include checks that are only concerned with the readability/maintainability of the code. The *IndConfig* was developed in an iterative manner based on the number of false positives generated.

3.2. Projects and context

We use two projects in order to evaluate the bug pattern tools in real project environments. Both analysed projects

[2]http://pmd.sourceforge.net/
[3]http://www.gimpel.com/html/pcl.htm
[4]http://www.klocwork.com
[5]http://www.gotdotnet.com/Team/FxCop/
[6]http://conqat.cs.tum.edu

were done at at Cirquent,[7] an international IT service and consulting company based in Germany. It offers services for the design, implementation, operation, and maintenance of business software systems in the domains production – especially automotive –, banking, insurance, and telecommunication. Cirquent has more than 1,500 employees working in several European countries.

Both projects are sales-support systems and typical examples of software developed in the Java Enterprise Environment (J2EE). For reasons of confidentiality we renamed the two projects to *project* X and *project* Y. *Project* X is a larger and mature system with almost 600,000 lines of code and 2,900 classes. It mainly contains a product configurator and customer management and has been in production since 2002. *Project* Y is smaller with about 40,000 lines of code and about 250 classes. It is also a sales-support system that is web-based and is used to create and manage product offers. The system has just been finished and entered system testing at the time of analysis. In both projects no formal inspections have been performed but the usual steps of unit, integration and system tests were in place.

3.3. Research questions

The aim of this study is to investigate in which ways bug pattern tools are best employed in the quality assurance process for software. Such tools can be used early in the development process directly after the code is written. Because early defect removal is cheaper than later removal [16] and because of the high degree of automation, the use of bug pattern tools is a chance for cost-reduction. We see three research questions that need to be answered in order to investigate whether this chance can be utilised.

RQ 1 *How many field defects need to be detected by the tools to be cost-efficient?*

Defect-detection techniques are used in the software development process to find and remove defects before they can manifest in field defect. Hence, it is necessary to compare the costs of defects during operation with the costs caused by the usage of the tools. Based on the assumption that defect removal in the field is more expensive than in-house, the number of defects to be detected for cost-efficiency can be analysed.

RQ 2 *Can bug pattern tools detect field defects or faults detected by other defect-detection techniques?*

Having established the necessary number of defects, it needs to be investigated whether bug pattern tools are effective in detecting defects that later would occur in the field. Moreover, assuming that removal in later phases is also expensive, it should be analysed whether defects are found that would also be detected with other defect-detection techniques.

[7]At the time of the study Cirquent was called Softlab

RQ 3 *Can fault-prone classes be identified using the tools?*

Finally, bug pattern tools do not need to detect defects directly to be efficient. They can also be valuable by identifying classes that likely contain such defects. This identification method can be based on the density of warnings from the tools for classes.

3.4. Study design: cost-efficiency

For the analysis of the cost-efficiency, we use our model of quality economics for defect-detection [17] as basis. It structures the costs of quality in different costs types as shown in Fig. 1.

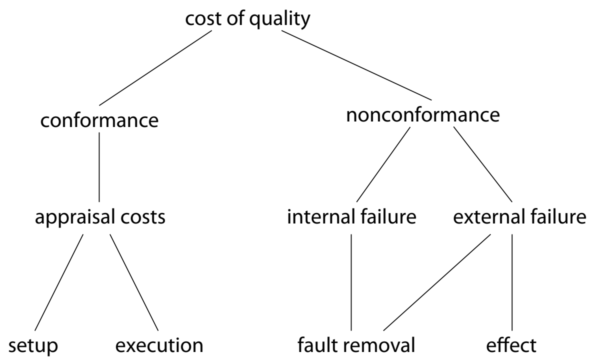

Figure 1. The structure of the cost types

Direct costs The appraisal costs comprise the main costs that are specific for a defect-detection technique. We detail those costs especially for bug pattern tools. The setup costs for the tools contain (1) tool costs, (2) installation costs, and (3) configuration costs. The tool costs are the price of the software itself and possibly costs for maintenance contracts. The installation costs are the effort necessary to install the tool on a machine so that it is usable with its StdConfig. Finally, the configuration costs contain the effort for reading the documentation of the detectors and creating a suitable IndConfig for the situation. The latter effort can usually be reused over several projects.

The execution costs of the tools can be divided into (1) tool execution costs and (2) report analysis costs. In general, the tool execution costs are negligible because the execution does not need human attention. Even starting the execution can be automated. Hence, the report analysis costs are decisive.

Depending on the used defect-detection technique, there are different costs for removing the found defects [16], the internal fault removal costs. This is separate from the report analysis costs that incur by identifying whether the warning is truly a fault. The removal costs stem from actually changing the code. Hence, it is mainly important how many real faults are detected by the bug pattern tools. We call the setup, execution and internal fault removal costs combined also the *direct costs* [17] because they occur directly when applying the tools.

Saved costs The costs that should be saved by using defect-detection techniques are the external removal costs. They contain all costs that occur when a failure takes place during operation of the software. Cirquent has a 3-level support system for *project X*. Only the failures that come to the third level are of actual interest. Those failures have their cause in code faults. Others might stem from wrong usage or configuration. When those costs can be avoided by a defect-detection technique, we call them *saved costs* [17].

Break even point We use the following equation to calculate the break even points for *project X* and *project Y* under the assumption that Y will have a similar support process as X.

$$\text{break even} = \frac{\text{direct costs}}{\text{saved costs per defect}} \qquad (1)$$

Hence, the break even point gives us the answer to RQ1: How many defects need to be detected in order to be cost-efficient?

3.5. Study design: effectiveness

We use two analysis approaches: (1) changes in different versions and (2) documented field defects. In the first approach, we analyse several versions of the systems and compare the warnings. This allows a comprehensive analysis of all the warnings of the tools and also of undocumented changes. This actually analyses whether defects were detected by other techniques like unit tests or reviews. The second approach analyses defects that occurred in the field. It constitutes the "hard" test whether it would actually have been possible to avoid a field defect by using a bug pattern tool. In combination, these two approaches can give us a complete picture of the defect-detection capabilities.

Different versions approach A comprehensive evaluation is possible by analysing several consecutive versions of the software (released and internal). This way, we see changes in the warnings and can retrace the reasons for the change by analysing the code changes. This involves three steps: (1) applying the tool to two consecutive versions, (2) comparing the warnings, and (3) analysing why certain anomalies appear in the old version and not in the new version.

For further analysis, the reasons for the discrepancies are classified in four categories: (1) documented failure: the anomaly caused a documented failure, (2) other code change: change without a documented failure, (3) tool error: the warning was incorrect, or (4) unknown: it was not possible to retrace the cause. This categorisation is based on the comments provided when the code is stored in the version management systems. If available, the change management systems and the original developers are consulted to confirm the categorisation. In summary, this approach answers part of RQ2: Can the tools detect defects that are detected by other techniques?

Field defects approach The second approach uses the field defect database and traces back these defects to code changes. Then it is analysed whether those code pieces could have been identified by tool warnings. This approach has also three steps: (1) extracting the documented field failures from the database, (2) analysing the failure cause in the corresponding version of the source code, and (3) checking the failure cause using the tools. One problem, besides empty commit comments, is that the tool FindBugs analyses the byte code and hence we need to recompile the whole software version which can be elaborate depending on the version control system used. This approach yields the second part of RQ2: Can the tools detect field defects?

3.6. Study design: fault-proneness

We also want to analyse if the tools can help to identify pieces of source code that are fault-prone. This is useful for focusing inspection and test activities. We therefore examine if classes with warnings from different tools are more likely to be changed during a bug fix than classes with no warnings or warnings from only one tool.

For this, we use our quality control framework CONQAT [5] to correlate results generated from both tools and thereby identify classes that contain (1) *no* warning, (2) warnings generated by *one* of the tools and (3) warnings from *both* tools. We then identify which classes were changed during a removal. For this analysis we defined a class to be fault-prone if it was changed during the correction of *more* than one defect. This answers RQ3: Can fault-prone classes be identified by the tools?

3.7. Threats to validity

Internal validity Because the analyses are all after-the-fact, no effects caused by instrumentation or other influences by us should have taken place. Because of effort limitation we do not analyse all field defects but select a subset randomly. From our experience with the kind of defects reported, we believe that the subset is representative but it is a threat to the validity of the study.

External validity Both studies were conducted at the same company. Hence, similar processes and guidelines in software development were used. Moreover, only two different tools were used on two different projects. Because *FindBugs* and *PMD* are open-source, well-known and widely used, we believe that they are representative for bug pattern tools in Java. However, results may be different with commercial tools because they might put different emphasis on specific detection capabilities. Only two tools were chosen because of the high effort for individual configurations. The results can only partly be transfered to other programming languages as there are other bug patterns. For example, bug patterns related to pointer arithmetics are not relevant for Java but are important for C. Furthermore, the results might not be valid for other static analyses that use more sophisticated means to detect anomalies in software.

4. Cost-efficiency results

For the two projects, the calculated internal costs for a developer hour are 60 Euro including all additional costs. For all analyses, we use the two different configurations described above: (1) the standard configuration of the tools (*StdConfig*) and (2) an individual configuration (*IndConfig*) that uses only a subset of the available bug patterns.

Configuration costs The effort needed for configuring a tool varies strongly. If the StdConfig of the tool is used, the effort is small. An IndConfig can require considerable effort that depends on the quality of the documentation, preciseness of the detector analysis, and the total number of detectors. We analyse the needed efforts for *FindBugs* and *PMD* from the case studies. All of the about 200 detectors in each tool were evaluated for the particular domain and purpose. A problem that arises there is that some of the detectors cannot be separated in the tool. For example, the detectors *possible null dereference* and *redundant null check* are combined in one detector although one hints at a fault whereas the other aims at improving the readability of the code. The usefulness of a detector cannot be determined by only reading the documentation but needs also be run and evaluated against real code in order to reduce false positives. This results in an average effort of 2 minutes per detector. With 200 detectors in a tool, this amounts to about 7 hours or 420 Euro.

Report analysis costs The report analysis costs depend strongly on the number of warnings produced by the tool which in turn depends on the configuration. A small, very focused configuration limits the number of warnings and hence the effort to analyse them. We discussed above that creating the configuration can be elaborate but it can avoid the analysis of many false positives. Moreover, we observed that the reporting of the warnings has a large influence on the analysis cost. Both tools can be integrated into the development environment *Eclipse*[8]. Thereby it is possible to link the warnings directly with the code in the editor. This simplifies analysis and removal. A report in a separate tool or in an HTML format creates a much higher hurdle. Furthermore, the size of the project, i. e. its number of lines of code, is important. The experience shows that on average the needed effort to analyse a single warning is 1 minute. In the StdConfig, *Project X* had 2,240 warnings that correspond to 2,240 Euro. The IndConfig produced only 178 warnings that cost 178 Euro. *Project Y* had 402 (StdConfig) and 141 (IndConfig) warnings with the corresponding costs of 402 Euro and 141 Euro.

Internal removal costs The effort for the change of the software is small because the tools explain the direct cause of the anomaly. The experience in the case studies was that on average it took 1 minute to remove a fault based on a tool warning. In *project X*, we found by consultation with the developers that 155 of the warnings are real faults that should be removed. Hence, the removal costs are 155 Euro. In *project Y*, the removal costs are 122 Euro.

External removal costs The third-level support needs an average effort of 3.3 hours per defect. In addition, the second-level support invested .5 hours before forwarding it to the third-level support. Hence, an average defect causes an effort of 3.8 hours which correspond to 228 Euro. Further failure effect costs are not considered in detail because they vary strongly over the various defects. However, we want to note that severe defects can cause unscheduled patches and releases that amount on average to over 2,000 Euro. We do not have data for the external removal costs of *project Y* yet.

Break even point A summary of the economic analyses including break even points for the two projects and configurations is shown in Tab. 1. All costs are given in Euro. It shows first the governing direct costs of applying bug pattern tools in a standard configuration and an individual configuration for each project. The potentially saved costs per defect are given where the costs for *project Y* are assumed to be similar to the costs in *X*. Using Eq. 1, we calculated the average break even point. The *StdConfig* in *project X* needs to find the most – 15 – potential field defects. The

[8]http://www.eclipse.org/

Table 1. Summary of the costs and break even points

Cost type	Project X		Project Y	
	StdConfig	IndConfig	StdConfig	IndConfig
Configuration costs	0	420	0	420
Report analysis costs	2,240	178	402	141
Internal removal costs	1,120	155	201	122
Saved costs (per defect)	228	228	228	228
Break even (average)	14.7	3.3	2.7	3.0
Saved costs (severe defect)	2,388	2,388	2,388	2,388
Break even (severe defect)	1.4	0.3	0.3	0.3

IndConfig is more realistic with 4 defects. In *project Y* both configurations seem to be able to be efficient. Finally, the costs for a severe defect as discussed above are given. In the case that the tools find a severe defect, they need to find in nearly all cases only a single one to be cost-efficient. Hence, for research question RQ1, the tools only need to detect a single severe defect or 3–15 normal defects in order to be cost-efficient.

5. Effectiveness results

Having investigated the number of defects that need to be detected in order to be cost-efficient, it is now important how many defects are actually found. We apply both analysis approaches to project X due to the existence of a well-kept database of field defects. For project Y we had to limit our analysis to the different versions approach as the project is not yet in production and therefore there is no database of field defects yet.

5.1. Project X

Different versions approach To apply this analysis approach, the bug pattern tools were run on the five versions a–e of project X with the *IndConfig* configuration set. A version can be internal or external. These five versions were developed within a time period of 2.5 years. For each new version we examined the set of warnings generated by the tools and analysed which warnings were present in the previous version but not in the current one.

The 600,000 lines of code of project X consist of the application's client, server and test code. Our analysis excluded the test code and focused on the server code from version b on. Unfortunately, we were not able to distinguish between client and server modules in version a as a package-level separation was introduced only in version b. In the four comparison steps we found a total number of 67 warnings that were removed from one version to the next. These removed warnings were analysed and categorised according to the schema described in Sec. 3.5. The results of this approach are shown in Tab. 2.

Table 2. Removed warnings in project X

Versions		Fault	Change	Tool	Unknown	Σ
a	b	0	11	9	2	22
b	c	0	22	3	2	27
c	d	0	8	4	0	12
d	e	0	5	1	0	6
Σ		0	46	17	4	67
%		0.0	68.6	25.4	6.0	100.0

The most important result of the analysis is shown in column *Fault* that illustrates that none of the warnings was removed because it actually was a fault that lead to a failure. One can see that the majority of warnings (46) were removed from one version to the next due to code changes that were not directly related to the warning. Examples are a refactoring that moved utility methods to a central location and a migration from Java version 1.3 to 1.4. Eleven of these warnings were removed during a bug fix but did, however, not describe the actual cause of the defect. Yet, for some faults an undocumented defect-detection technique, such as a peer review, can be the reason for the removal.

The second biggest group of removed warnings are tool errors, i. e. warnings that were not reported anymore although they were not fixed. Reasons for this are minor refactorings (e. g. extract method) that do not actually fix the problem but make it *invisible* for the tool. In four cases we were not able to figure out why the warning was removed as the corresponding commit messages were either empty or did not contain enough information.

Field defects approach To determine how many of the defects that really occurred in the field could have been found by using bug pattern tools we use our other analysis approach. We analyse a four-year period in the bug tracking database and randomly select a subset of the failures. For each failure we analyse its cause in the corresponding source code version and check if the cause could have been identified by one of the tools. The bug tracking database contained 615 reports of which we randomly selected 99 reports (16.1%). Of the 99 reports we had to discard 27 as their resolution could not be linked to particular locations in the source code. For each of the remaining 72 reports we retrieved the corresponding source code from the version management system, analysed the fault that caused the defect and checked if it would have been indicated by one of the bug pattern tools.

Interestingly, we could not find a *single* case where a field defect could be related to a warning generated by one of the bug pattern tools. We see the reasons for this in the inherently limited power of the bug patterns used in the tools as they mainly work on a syntactical level. We illustrate different types of defects and explain why those bug pattern tools are not able to detect them.

The majority of the defects (39) were caused by logical faults that fully elude static analysis. Examples are incomplete case differentiations and calls to wrong API methods. Another group of defects (8) that can hardly be found by static analysis, are defects caused by improperly chosen constant values. Examples are font size and paper size for the print functionality. A rather large group of the defects is purely user interface related. Examples are improper distances between UI elements and inappropriate screen layouts or the flawed display of the application's logo on a particular operating system version. We found that the UI related defects could be divided in a group of defects caused by logical faults and a group that we suspect to be identifiable by special bug patterns. An example is a user customisation mode of the application that requires every UI element to be visible to be customised. We can imagine a special bug patterns that checks if the corresponding `setVisible()` method is called at least once for every UI element. However, we did not evaluate the application of specialised bug patterns in this study.

Discussion The results of both analysis approaches suggest that commonly used bug pattern tools are not well suited for finding field defects. As pointed out we believe that this is mainly due to the tools' inherent inability to find logical defects as they purely focus on the source (or byte) code of the system under investigation. As research in reverse engineering shows, the source code is a relatively poor source of information with respect to questions of this kind [3, 6] and does not encode the information to detect such defects.

An additional reason for the unsuccessful application of the tools in this study may be the relatively late application of the tools. Project X was already in production for four years and reached a high level of maturity during this time. As no bug database was available for initial development, test and early operation phase we cannot say if the application of the tools would have been more effective in earlier phases of the project's life cycle.

5.2. Project Y

To investigate the influence of the project life cycle phase on our results we conducted a similar study for project Y. As no bug tracking database was used in this early development phase, we had to limit our analyses to the different versions approach where we used the analysis tools to compare different versions of the code. We analysed six versions a–f that were developed in a period of two months. Again, we used our custom tool configuration *IndConfig*.

In five comparison steps we found a total number of 24 warnings that were removed from one version to the other. We analysed and categorised them by examining the commit messages and occasionally interviewing the developers. Due to their support we could exclude the category *unknown* for this analysis. The results of the study are shown in Tab. 3.

Table 3. Removed warnings in project Y

Versions		Fault	Change	Tool	Σ
a	b	1	2	5	8
b	c	3	10	0	13
c	d	0	0	0	0
d	e	0	3	0	3
e	f	0	0	0	0
Σ		4	15	5	24
%		16.7	62.5	20.8	100.0

Most importantly, this time, we found 4 warnings that were removed because they actually caused a defect. Two of these warnings related to database connections that were opened but never closed. The other two concerned possible null pointer dereferences. Hence, there is a small overlap to other defect-detection techniques.

Again, the most frequent reason for the removal of warnings were changes to the code that did not relate to the warning itself. This included refactorings as well as deletion of the pieces of code that raised the warning.

Similar to our experience with project X we found that a number of warnings was not reported due to deficiencies of the tools. An example is the change from `x.equals("")` to `x.trim().equals("")`. Here PMD suggests to reverse the former expression to avoid a possible null pointer dereference, but does not complain about the latter expression that exhibits the same risk.

5.3. Discussion

Our study results suggest that bug pattern tools do generally not fulfil what some of their manufacturers' website claim. Due to their syntactic nature they are limited to finding simple patterns and miss many kinds of logical bugs. Unfortunately, we found the latter kind of defects to dominate in a system that reached a certain degree of maturity. For a project in earlier development phase the bug pattern tools were able to identify a small number of faults. However, it is not clear if these faults would have caused a field defect later on. Hence, we need to answer RQ2 rather negatively. There is only a small number of defects detected by other techniques and in the field that are detectable by bug pattern tools.

Furthermore, we found that bug pattern tools do not only create a large number of false positives but do also miss patterns that we expected them to identify correctly. We found that bug pattern tools in general need to be configured carefully in order to keep the total number of generated warnings in check.

6. Fault-proneness results

Although we did not find field defects that the bug pattern tools could detect directly, we analysed if they can help to identify pieces of source code that are fault-prone (RQ3). To achieve this we classified the classes of project X into the categories *None, One* and *Both* which means that neither tool, one of the tools or both tools raised a warning. The classification for project X is shown in Tab. 4. Note that we excluded version a from this analysis as we limited our analysis to the client component of the application. Before version b the separation of client and server modules was not reflected in the package structure of the program and we could therefore not distinguish between the modules. For all analyses we used the custom tool configuration *IndConf.*

Table 4. Analysis results for project X

V.	# Classes	LOC	None	One	Both
b	689	151,913	77.4%	18.4%	4.2%
c	764	168,729	77.9%	18.2%	3.9%
d	774	175,534	77.4%	18.3%	4.3%
e	954	227,201	76.1%	19.3%	4.6%

For investigating in how far the presence of warnings for a class is an indicator for fault-proneness we examined which classes were changed in Version e during the correction of the 72 analysed defects. For each corrected class we determined to which of the categories *None, One* and *Both* it belongs.

Overall 28 classes were changed during the correction of the 72 bugs. Classes were changed at most 5 times and no class was changed only once. For each category Table 5, column Σ gives the number of classes in this category and the columns *1x–5x* show how many classes of a category were changed 1 to 5 times.

Table 5. Fault-proneness categorisation

Category	1x	2x	3x	4x	5x	Σ
Both	0	2	2	2	2	8
One	0	7	3	4	5	19
None	0	1	0	0	0	1
Σ	0	10	5	6	7	28

To answer RQ 3 we analyse if there is a correlation between the category membership (*None, One, Both*) and the number of times a class is changed during bug correction. We use the Spearman rank correlation as it can be applied for ordinal scales and does not require a linear relation between the variables.

For the 954 observations we find a positive correlation coefficient of 0.34 with a 2-tailed p-value < 0.0001. This result indicates that classes for which both tools raised warnings are more likely to "participate" in a bug than classes that are annotated by only one tool. These classes are again more likely to be fault-prone than classes without warnings raised by either of the tools.

It is not clear yet how far these results can be generalised as they largely depend on the selected tools and their configurations. However, the results indicate quite strongly that bug pattern tools can be used to find system "hot-spots" that are prone to contain faults. Additionally, these results reinforce the findings on the benefit of using multiple analysis tools in combination.

7. Discussion & consequences

The experiences from the case studies and the results discussed above lead to several methodological consequences w.r.t. the use of bug pattern tools in the development of software systems. Automation in general is a key to more efficiency and we believe that bug pattern tools can contribute to this automation. However, it is still not completely understood when and how to use the tools effectively and efficiently. We derive a set of guidelines to improve this situation:

- *Use the tools often and early in the development to keep the number of warnings small.* This helps to exploit learning effects of the developers that prevent introducing defects by avoiding code that produces warnings. Moreover, the psychological barrier to even start on working on a large number of warnings is high. The repeated use of the tools avoids this problem.

- *Use different tools in combination.* The experience here and in other studies [15, 18] shows that there is only a small subset of similar detectors in the various tools. A combination can fully use the potential of detecting bug patterns.
- *Create an individual configuration for each tool.* One clear result from the case studies is that this can largely reduce the false positives produced by the tools. Thereby, the efficiency is increased because less warnings need to be analysed and the acceptance raises because the reports are more useful to the developers. Moreover, the effort of creating these configuration is spent not only for one projects but the configuration can be reused.
- *Integrate the warnings into your IDE.* The tools should be able to annotate the warnings directly in the text editor that you use and allow also a direct link to the description of the warning. This fosters a quick and easy fault removal and also avoids a psychological barrier. The need to change from the tool to the text editor might keep the developers from working with the tools.
- *Accumulate and analyse the results from the tools into a report.* This can be done most efficiently using a quality report tool. Tools such as CONQAT [5] allow an easy integration of different tools to simple reports. The reports can be used for status overviews or trend analyses. Moreover, the identification of fault-prone components (cf. Sec. 6) can be used here to focus further quality assurance.

8. Related work

We analysed in [18] several systems at the mobile network provider O_2 using three different bug pattern tools for Java. One aim was to investigate the classes of defects found and a comparison with reviews and tests. The main results were that tests detect different defect types but reviews similar types as bug pattern tools. Furthermore, it was identified that the tools have high false positive rates and that this hampers the use of the tools in practice.

Rutar, Almazan and Foster [15] compared several bug finding tools for Java. In detail, they used five different tools on five rather large open source software packages. The results were that the tools generate only a small overlap in the warning for a software. They concluded, similar to our findings, that different tools should be used in combination. Finally, they also found that the amount of information produced is problematic but they did not use individual configurations.

Hovemeyer [10] analysed FindBugs in commercial and student projects. He found that the false positive rates of different detectors in FindBugs vary strongly. This supports our finding that an individual configuration can reduce the false positives. Moreover, he conducted an informal survey with FindBugs users. One observation there supports our fault-proneness approach: “code which confuses static analysis is also likely to confuse developers trying to understand it.” Also the learning effect of bug pattern tools is confirmed in the survey. Ayewah et al. [2] continued the application of FindBugs on industrial strength software. They found defects in production software that was severe and was later removed. This supports our conjecture that at least some field defects are detectable by bug pattern tools.

Zheng et al. [19] analysed three systems developed at Nortel Networks. They also looked into the economical implications of using static analysis tools. However, the main difference is that the software was written in C/C++ that allows a whole class of additional defects, for example misusing pointer arithmetic. Nevertheless, they had conclusions similar to our findings. Firstly, they also confirmed that the results of static analysis can be used to detect fault-prone components. Secondly, the costs of a detected fault are on average comparable to a fault detected by manual inspection. However, it was not analysed how many field defects need to be found by a tool in order to reach the break even point. Finally, they also observed that the defects reported in the field are mainly of a different class than the defects reported by the tools. However, Reimer et al. [14] showed that domain-specific bug patterns can be very effective to this end.

Nagappan and Ball [13] found that the density of warnings of their PREfix and PREfast tools can be used to predict pre-release defect density.

9. Conclusions

We analysed two widely-used bug pattern tools for Java in two industrial projects in order to evaluate their use in defect-detection. The main aim was to investigate whether bug pattern tools are able to detect defects that would occur in the field. The study could not confirm that this is possible. However, because it contains only two case studies a more detailed evaluation is needed. We believe that assuming that a few field defects can be found is still realistic. Therefore, we analysed the break even point – the number of defects needed to detect so that the tools avoid more costs than they incur. We found that about 3–4 of such defects are needed. However, using bug pattern tools has more effects. It can reduce efforts in later inspections and tests and also supports the continuous learning of the developers. This way the introduction of defects can be prevented.

We see the application in the wider spectrum of quality assurance of software. We specifically work on quality models for software [7] that also support automatic analysis of certain aspects of the model. Bug pattern tools are one way to achieve that, especially when they are extensible. In this context, our experience was that it pays off most of the time to automate such analyses. In any case, the output of the tools is a valuable part of analysing quality.

Several further questions arise from our work. It would be interesting to investigate the limits of static analysis and especially bug patterns. When is static analysis possible, when does it make sense? When do we need dynamic tests? Can we generate "generic" dynamic tests for specific bug patterns that cannot be checked statically? When is it worthwhile to develop custom bug patterns for problems that occur repeatedly? How could the development of such custom bug patterns be supported in an efficient manner (cf. [4])? Moreover, for some of the bug patterns it might be possible to automatically correct them. To what extent is this possible? Is this really useful because the learning effect is destroyed? Finally, it is necessary that further studies confirm our findings. In particular, it is important to note that we concentrate on the effects of the use of bug pattern on the reliability of a system. Specific anomalies, however, may have a strong influence on the maintainability, security, or safety of a software. These aspects have not been specifically investigated by this study. Nevertheless, our work in combination with the related work is already able to give insights into various aspects of static analysis tools in practice.

References

[1] E. Allen. *Bug Patterns in Java*. Computer Bookshops, 2002.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proc. 7th Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, pages 1–8. ACM Press, 2007.

[3] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[4] B. Chelf, D. Engler, and S. Hallem. How to write system-specific, static checkers in Metal. In *Proc. 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '02)*, pages 51–60, 2002.

[5] F. Deissenboeck, M. Pizka, and T. Seifert. Tool Support for Continuous Quality Assessment. In *Proc. 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP '05)*, pages 127–136. IEEE Computer Society Press, 2005.

[6] F. Deissenboeck and D. Ratiu. A unified meta-model for concept-based reverse engineering. In *Proc. 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering (ATEM '06)*. Johannes Gutenberg-Universität Mainz, 2006.

[7] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An activity-based quality model for maintainability. In *Proc. 23rd International Conference on Software Maintenance (ICSM '07)*. IEEE Computer Society Press, 2007.

[8] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] D. Hovemeyer. *Simple and Effective Static Analysis to Find Bugs*. PhD dissertation, University of Maryland, 2005.

[11] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[12] IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[13] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proc. 27th International Conference on Software Engineering (ICSE '05)*, pages 580–586. IEEE CS Press, 2005.

[14] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. SABER: Smart analysis based error reduction. In *Proc. International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 243–251. ACM Press, 2004.

[15] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proc. 15th IEEE International Symposium on Software Reliability Engineering (ISSRE '04)*, pages 245–256. IEEE CS Press, 2004.

[16] S. Wagner. A literature survey of the quality economics of defect-detection techniques. In *Proc. 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE '06)*. ACM Press, 2006.

[17] S. Wagner. A model and sensitivity analysis of the quality economics of defect-detection techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA '06)*, pages 73–83. ACM Press, 2006.

[18] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th International Conference on Testing of Communicating Systems (TestCom '05)*, volume 3502 of *LNCS*, pages 40–55. Springer, 2005.

[19] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.