Research Proposal
Detecting Test Bugs Using Static Analysis Tools

Kevin van den Bekerom
kvandenbekerom@sig.eu

Q1 2016, 17 pages

Supervisor: dr. Magiel Bruntink, m.bruntink@sig.eu
Host organisation: Software Improvement Group, https://www.sig.eu/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Chapter 1

Research Proposal

Suggestions by Barbara — Questions

1. Make RQ3 the goal of the research?
2. How can static source code analysis tools find test bugs? Make subquestions.
3. For example: What analysis techniques are required to find test bugs? I can than look per category (or a subset of) how I theoretically would look for the bug. Dataflow analysis? Callgraph analysis? Simple regular expressions? Constructing full state machines? etc. How would I do this? Is this feasible? Or as a proof of concept try to implement one test bug in FindBugs, is this sufficient?
4. I listed some questions for RQ2 that I want to look at. Barbara correctly noted that this is infeasible. My purpose is not in answering these questions, but using them as guidelines to answer RQ2. How should I make this explicit?
5. For RQ1, leave out the specifics and mention them later. *What test bugs as categorized by Vahabzadeh et al.can static source code analysis tools find?* I will then do a case study and later argue why the results can be (to a certain extent) be generalized.
6. Why only Java tools? Java most occurring programming language. Most clients from SIG Java.

1.1 Project summary

Developers write test cases to test behavior of production code. Test cases are themselves code. It happens that there are errors in test code. A test fails but does not signal a bug in the production code. Vahabzadeh et al. studied and classified so-called *test bugs* [14]. A test bug is a defect in test code that does not signal a defect in production code (false positive) or should have signaled a defect in production code (silent horror).

According to Vahabzadeh et al., FindBugs, a popular bug finding tool for Java, could not find any test bugs as classified by them in the same paper [14]. My goal is to investigate the current ability of tools to help developers find test bugs. For this I will use the classification of test bugs made by Vahabzadeh et al. I will replicate their research on FindBugs, where they examined what test bugs FindBugs could find. I will use this approach to set up my research in order to investigate other popular bug finding tools; PMD and JLint. In the first phase I want to answer the following question:

RQ1: What test bugs, as described by Vahabzadeh et al. can popular Java bug finding tools [1] find?

[1] FindBugs, PMD, JLint

After this research phase I will have a list of test bugs that *can* be found by the investigated tools. I will use the list of test bug subcategories ad defined by Vahabzadeh et al. to compute a list of undetected test bugs. In the same paper the percentage of test bugs for each subcategory is given. For the category having the most test bugs I will implement - as a proof of concept - a bug detector in FindBugs. After the second phase I want to have answered the following research question:

RQ2: How can a static source code analyzing tool be extended to find test bugs?

This will result in a proof of concept, where I try to implement a bug detector for one of the test bug categories described by Vahabzadeh et al. in FindBugs. I also want to assess the usefulness of the implemented detector. For my research to be relevant, developers should be able to effectively use the bug detector. Relevant questions at this point are:

- What static source code analysis techniques are available?
- What are the limitations of these techniques?
- What type of analysis is not possible when only considering static source code?
- For a given test bug category: what analysis techniques are required to find the bug?

In answering these questions I will find what (if any) test bug can be implemented using static source code analysis. Note that the focus is on finding the test bug first. Reaching an acceptable false positive rate, or fast analysis speed will be investigated later, resulting in the following research question:

RQ3: Can static source code analysis tools help developers find test bugs?

This question is composed of three subquestions:

RQ2.1 Does the bug detector have an acceptable false positive rate?
See the hypothesis for details (1.3)

RQ2.2 Does the bug detector's speed have a negative effect on usage by developers?

RQ2.3 Does the bug report give developers enough information to localize and solve the bug?

1.2 Problem analysis

Vahabzadeh et al. found that a popular bug detection tool for Java, FindBugs, could not detect test bugs. They did not investigate whether other tools could find test bugs. Usually, when a test fails the developer has to fix the error or the build will fail. Can we add tool support for detecting test bugs that are usable for developers? At SIG, test bugs do occur. They can benefit from tool support for detecting test bugs, which mostly manifest themselves as false positive test cases.

1.2.1 Tool support for finding (test) bugs

FindBugs was initially introduced by Hovemeyer et al. [8] and compared with other popular bug finding tools for Java by Rutar et al. [13]. Ayewah et al. investigated the real world usage of FindBugs. They showed that developers readily check errors with high risk, and the usage of tools like FindBugs works best when integrated into the development process [2].

1.2.2 Detecting test bugs

There exists some research on finding errors in unit and integration tests. Greiler et al. looked at test fixtures [5]. A test fixture is a way to setup a (unit) test, e.g. bringing the system into

the correct state, accessing resources, etc. They mainly investigated test fixture smells, and found their occurrence in practice. A fixture smell is a symptom of the test bugs in the category *Resource Handling*, described by Vahabzadeh et al. Herzig et al. investigated false test alarms in general [7]. They focused on integration testing and used machine learning, in particular association rules, to classify test case failures as false alarms. Luo et al. investigated flaky test cases, which are tests that fail due to concurrency issues [10]. They identified the most occurring categories of flaky tests (async wait, concurrency, test order dependency) and explained how flaky tests manifest themselves. They also investigated strategies developers use to fix flaky tests. Hao et al. proposed a machine learning approach adopting the Best-First Decision Tree algorithm to classify test failures as being caused by the code under test or the test code itself [6]. They reached high accuracy (around 90%) when the training set stems from a previous version from the same program. Accuracy deteriorates (60%) when training data comes from a different program. DiGiuseppe et al. proposed a novel way to cluster program failures based on semantics [4]. The method they used proved to be more accurate than previous cluster-based approaches and can guide an approach to incorporate the semantic information of issue reports into failure detection.

1.2.3 (Test) Code smells

Khomh et al. implemented a detection method using Bayesian Belief Networks (BBN) [9]. They showed that a BBN can be used to prioritize design issues, giving probabilities to classes potentially having a design smell. Their approach can be used to validate scientific research where an initial classification has been made, and therefore can be used as training set. Marinescu proposed a systematic way to detect design flaws. Based on data filters and composition filters (boolean and set operators) symptoms and detection rules for a design smell can be transformed into a detection pattern, specifically aimed at finding a particular design smell. Bavota et al. studied the diffusion of test smells among classes. They found that a high percentage of classes had at least one test smell. They also found that test smells negatively affect maintenance.

1.2.4 Testing in general

Athanasiou et al. designed a model for measuring the quality of test code [1]. They incorporated static source code metrics to assess the quality of source code based on completeness, effectiveness and maintainability. Their work makes it possible to study the behavior between test code quality and test bugs. It would be interesting to find out if a correlation exists between test quality and (number of) test bugs. Zaidman et al. studied the co-evolution of production and test code [16]. Their research could be a starting point to study the co-evolution of production and test bugs.

1.3 Research method

Research difficulties

- Original paper by Vahabzadeh et al., on which my research is based, is very recent. Not much is known about test bugs.
- The tool, that Vahabzadeh et al. used to gather data from JIRA repositories does not work out of the box.
- Obtaining data from a representative population of developers that (regularly) solve test bugs (to answer RQ3.3).

Sources usage and documenting these.

- *Authors of the Test Bugs paper*: They can help me setup their tool in my environment. I can use their tool when answering RQ1. In involving the authors in my research I hope to get them excited to think along. They might also warn me for pitfalls they ran into, which may result in time savings.

- *Papers*: first assess relevance. If relevant, then read the paper and make a quick summary. Keep summaries in one folder. Also include relevance in the same file as the summary. Update bibliography (Bibtex) to include reference. This can be used to quickly see what papers I have already read.
- *Internet sources*: Mainly use the internet to find tutorials about several subject, e.g. making a new detector for FindBugs. Only use these sources to come to a better understanding of a subject.
- *Colleagues at SIG*. Make research subject public and debatable. Involve colleagues in my subject to get feedback from different people with different backgrounds. Incorporate feedback either in official document (Research Proposal, Thesis) or make a digital note. Every week there are research meetings to discuss the progress I made and discuss any issues I currently have.
- *Source code*: I have to investigate test bugs of software systems which involves looking at source code.
- *Issue Tracking System (ITS) issues*: test bugs can be found by looking at issues in the ITS. I will use these issues, combined with looking at the source code to understand why an issue is a test bug, and why a certain category of test bug.
- *Data from previous research by Vahabzadeh et al.*: They found more than 5000 test bugs by mining ITS and VCS systems. Online I can find the identifier of the issue. They also have a list of issue identifiers with a corresponding bug category linked to it.

Experiments, research, proof of concept
I will do an experiment, similar to the one Vahabzadeh et al. did in their paper, where they investigated whether FindBugs could find a certain type of bug in real software. I will replicate their experiment, and try to improve it. Vahabzadeh et al. only investigated cases where there was a difference in reported bugs between two consecutive commits. They assumed that a reported bug by FindBugs will always disappear after a bugfix when the reported bug correctly predicts the bug in (test) code. It might be that a reported bug points in the general direction but the bug pattern is too coarse to detect the fine detail of a certain bug in test code. I will repeat the research for PMD and JLint. I will also look at the location each reported bug by FindBugs points to, ruling out all reported bugs that have nothing to do with the test bug. Based on manual inspection, I will judge whether the test bug could have been found with the (set of) reported bugs by FindBugs.

For the second phase 2, see 1.6.2. I will research what algorithms and/or techniques I can use to build a bug detector for (several) test bug categories. The result of this research will be input to the implementation phase, where I will implement one bug detector as a proof of concept. My goal is to show that test bugs can be found using static code analysis tools, in particular FindBugs.

What method will you use?
I will take a positivist stance and will apply the mixed method approach. To answer RQ1 1.1 I will do a case study where I look at three popular Java bug finding tools. To answer RQ2 I will do a literature study to find the limitations of static source code analysis as well as the requirements to detect certain bug patterns. To answer RQ3 I will use again a quantitative analysis (hypothesis testing) (RQ3.1) combined with a qualitative analysis (survey) (RQ3.3). A restricting factor for the survey is the willingness of developers to sacrifice some of their time to help in the survey. Graduate and undergraduate students are ill-suited to act as representative population since they do not have the expertise to solve the (often) complex test bugs. Another aspect to take into consideration is the experience developers have with solving test bugs. When a developer has never solved a test bug, it might be hard for him to pose fixes, even based on a bug report.

Hypotheses

RQ1 : H_0 : *The set of studied state-of-the-art bug detection tools can jointly find less than 25% of the test bugs identified by Vahabzadeh et al. [14].*

RQ2 : H_0 : *The implemented bug detector finds bugs with high accuracy (false negative rate below 20 %).*

RQ3.1 : H_0 : *The developed bug detector has a false positive rate of less than 50%.*

Validation

RQ1 : I will evaluate RQ1 by applying the same method used by Vahabzadeh et al. in their study. Instead of using only one tool (FindBugs) I will use several tools (FindBugs, JLint, PMD). In using multiple tools, diverse in their way of measuring, the external validity will increase, i.e. confidence in generalizing will increase. Tools from other programming languages are not covered however. I will need to find out what makes those tools different from the Java tools in order to generalize my results for all programming languages.

RQ2 : A working bug detector in FindBugs having a convincing low percentage of false negatives, i.e. missed bugs.

RQ3.1 : I will use the method developed in phase 1, see section 1.6.1, but turn off all detectors, except the newly developed one. To see if the developed detector actually finds bugs, I will use the online data by Vahabzadeh et al. They detected several test bugs in different categories. I will use all projects that I can run, for which Vahabzadeh et al. have identified a bug in the category the detector should find. I divide this number by the number of reported bugs by BugFinder to obtain the false positive rate.

RQ3.2 : FindBugs has three speed categories: fast, moderate and slow. If the bug detector operates in the moderate to fast category, the detector is deemed fast enough. If the detector operates in the slow category, I will compare the running speed with all other detectors in this category and based on its ranking determine if the bug detector is fast enough. It will depend on what techniques the bug detector is using and the severity of the bug to determine if it is worth the running time.

RQ3.3 : Collect a number of real bugs and patches. Ask developers to indicate where and how they would solve a given bug, based on the bug report issued by my bug detector.

1.4 Expected results of the project

- For RQ1 I expect to find that less than 25 % of test bugs can be found by at least one investigated tool.
- For RQ2 I expect it to be possible to build a custom bug detector for FindBugs, for a test bug that occurs relatively often.
- For RQ3 I expect to have build an usable bug detector that developers want to use, and can find bugs in real world software systems.

1.5 Required expertise for this project

1. Statistical toolkit R: for some basic statistics for study in phase 1 1.6.1.
2. Using and running FindBugs, PMD, JLint.
3. Compiling open source programs.
4. Data mining GIT repositories if TestBugs tool is unavailable. I will need to dive into the jGit library.
5. Understanding the causes and reasons behind (test) bugs. If I want to write a detector for FindBugs, I will first need to know if it is possible to write a detector for a particular bug. I can only decide this if I have a good understanding what causes the bug to occur, and a general idea of how to detect the bug.

6. Working with Java Bytecode (FindBugs scans Java Bytecode).
7. Writing a plugin (custom detector) for FindBugs.

1.6 Timeline

1. Replicate approach of Vahabzadeh et al. for finding which test bugs FindBugs can find.
   (a) Build GIT repository mining tool. Should be able to get commit data based on keywords. I want to get the commit that solved the JIRA issue (test bug). I also want the commit *just* before the issue-solving commit.
   (b) Get all programs for which JIRA data is available (list of issue identifiers that point to test bugs can be found online) and work with a specific bug-finding tool.
   (c) Analyze bug reports after applying the tool to the program at the time of the problem-solving commit, and at the time of the commit just before that.
   (d) Whenever there is a difference in reported bugs, analyze if the reported bug could have helped a developer prevent the bug from occurring (Validation based on questionnaire??).
2. Extend above approach for PDM and JLint.
3. Build framework for normalizing bug reports, to be able to easily extend other tools in the research.
4. Make a list of tools linked to test bugs they can detect. Based on the paper by Vahabzadeh et al., make a list of bugs that can not yet be detected.
5. Implement a bug detector for FindBugs for a test bug in a category that is most occurring according to Vahabzadeh et al., which no tool can detect yet.
6. Validate approach using the data from Vahabzadeh et al. Sample all projects for which test bugs have been identified, and for which the right category of test bug has been identified. Then get all projects which can be analyzed with FindBugs. For those projects, apply method in (1) and check if FindBugs reports bugs that are useful for developers to prevent/solve the test bug. Also run some projects for which no test bugs have been identified to check for false positives.
7. If I have sufficient time left, I will repeat steps 5 and 6.

1.6.1 Phase 1

Step 1, 2, 3 and 4 constitute phase 1. Phase 1 will take around 1.5 months. In Phase 1 I will continiously work on writing my thesis. The focus is not on writing coherent stories, but collecting the data such that it can easily be rewritten. The focus in Phase 1 is on setting up the tooling research, which will again be used in Phase 2. Result of Phase 1 is a list bug pattern to implement in phase 2. I will choose one of them to start with. Furthermore I will know for each investigated tool what test bugs it can find.

1.6.2 Phase 2

The first month will be reserved for implementing and validating a bug detector in FindBugs. The last 2 weeks will be reserved for rewriting the thesis.

1.7 Risks

- I never worked with byte code. Implementing a bug detector in FindBugs is supposed to be easy [8]. This is true, once you have developed several bug patterns already. Learning how the different detectors work might take longer than expected. The complexity of the bug detector

I will implement is not known yet. It might be a fairly straightforward detector, or might be extremely complex. If a certain detectors proves too difficult to implement, I can choose the next test bug in the list.

- Data by Vahabzadeh et al. might not be up to date anymore. I have not yet received a response from the authors in my request for help with their tool. I have to work with the data they obtained.
- After having answered RQ1 1.1, it might turn out that every test bug can already be detected. In this case I can refocus my research on combining the results of different analysis techniques in a *test bug detector meta-tool*.
- It might turn out that none of the identified bug patterns after Phase 1, see section 1.6.1, can be implemented using static source code analysis techniques. I that case I will resort to machine learning and build a separate tool as a proof of concept. I will use the classified test bugs by Vahabzadeh et al. as training set to detect various test bugs.

Chapter 2

Literature Study

2.1 An Empirical Study of Bugs in Test Code

As of 2015, no study existed that categorized bugs in test code. Vahabzadeh et al. researched many open source projects, written in different programming languages to find out what categories of test bugs existed in test code [14]. A test bug is either a False Positive (test fails but no defect in production code), or Silent Horror (test passes, but should have detected a bug in production code).

Vahabzadeh et al. started looking for issues in Issue Tracking Systems (in particular JIRA). If these issues were later resolved, with a fix in test code, they classified the issue as a test bug. From the set of test bugs they obtained (5556) they sampled 443 cases which have undergone manual inspection. For false positives, the most prominent categories were semantic bugs and flaky tests. A flaky test sometimes fails due to concurrency issues. Silent horror tests (test passes although there is a bug in production code) contribute to only 3% of all test bugs.

Vahabzadeh et al. also researched the issue solving differences between test code and production code bugs. They found that test bugs were solved much faster than production code bugs.

Finally they investigated whether popular bug solving tools could find test bugs they categorized. For TestBugs, a popular bug finder for Java, they found that the tool could not find a single test bug.

When a classification of test bugs exists, further research can focus on automatically finding and classifying test bugs. Machine learning algorithms could be used for classification. The data set produced by the authors then serves as a training set. Another prominent research topic is relating test bugs to static source code analysis (metrics). What metric possibly predicts which test bug? It is not striking that current bug finding tools do not have the capabilities to find test bugs, given the fact that no classification existed prior to this work. Now such a classification exists, tools could be extended to also find (several) test bugs.

2.2 Test Code Quality and Its Relation to Issue Handling Performance

Athanasiou et al. studied the relation between test code quality and issue handling performance [1]. They constructed a quality framework for test code on three categories:

1. *completeness*: how complete is the system tested? Achieved using program slicing to statically estimate code coverage. Also the decision point coverage is estimated.
2. *effectiveness*: hoe effective is the test code in detecting and localization faults.
3. *maintainability*: how maintainable is the test code. They build a maintainability model, which is based on the maintainability model of SIG.

The test quality model was calibrated using benchmarking. They tested the quality model on two commercial software systems. The results were validated by two experienced consultants who knew a

lot about the system. The results were closely alligned, but the consultants generally rated the test code quality of a system higher.

To measure the relation between test code quality and issue resolution speed, a study was conducted on several open source projects. Athanasiou et al. found that test code quality has a significant correlation with productivity and throughput. Productivity was measured as the normalized number of resolved issues per month divided by the number of developers. Throughput was measured as the normalized number of resolved issues per month divided by the KLOC.

Issue resolution speed; time between assignment of issue to a developer, and resolving the issue, showed no significant correlation with test code quality. Issue resolution time is hard to measure due to several confounding factors:

- Hard to measure number of developers working on an issue. One developer can be assigned, but multiple work on the issue.
- A developer might not work constantly on the same issue.
- Issues get reopened, which adds resolution time. Issues not yet being reopened might not accurately reflect resolution time.
- The experience of the developer is not taken into consideration.

Their work can be used to measure test code quality and use these measurements for determining the relation between test bugs and test code quality. Athanasiou et al. incorporated several metrics in their quality model, so it might also be interesting to find out the relation of individual metrics on test bugs and vice versa, i.e. what is the relation between bug categories on test code quality?

2.3 Mining Software Repositories to Study Co-evolution of Production and Test Code

Zaidman et al. investigated various aspects of production and test code over time, called views. They visualized the different views using a dot chart with colors.

The change history view plots addition and modification of unit tests, and units under test. Unit tests and units under test are plotted on the same horizontal line. Each vertical represents a commit. A vertical bar of the same color can represent work on a lot of test cases.

The growth history view plots several size metrics (production lines of code, test lines of code, number of classes, etc.) of production and test code against each other to reveal growth patterns. Zaidman et al. use an arrow grid to summarize the different growth behaviors.

The test quality evolution view plots test code coverage against test code volume for minor and major releases. An increasing line in coverage represents good test health.

Zaidman et al. applied their model on two Java programs: Checkstyle and ArgoUML. They found periods of pure test code writing. They also found simultaneous introduction of test and production code. For Checkstyle, size metrics of test and production code followed each other closely. For ArgoUML, they witnessed a more staged approach, i.e. periods of low increase in test code size, followed by periods of high increase in test code size. No system showed a sharp increase in testing effort just before a major release. Test-Driven Development could be detected for Checkstyle.The also found that test code coverage grows alongside test code fraction.

Zaidman et al. studied the co-evolution of test and production code. It could be interesting to study the co-evolution of test and production code bugs. Does one class of production code bugs always follow after a certain bug in test code?

2.4 Automated Detection of Test Fixture Strategies and Smells

The Test Fixture smell, as presented by van Deursen [15] was extended into five additional Test Fixture smells [5]. A test fixture is a way to setup a (unit) test, e.g. bringing the system into the correct state, accessing resources, etc. An *inline-setup* occurs when a test method has all the setup

code inside the method. Helper methods can also contain the setup code, which is called *delegate* setup. When features of the jUnit framework (naming conventions, annotations) are used to mark setup methods, it is called *implicit setup*. Test Fixtures are:

- *General Fixture*: Broad functionality in implicit setup and tests only use part of that functionality.
- *Test Maverick*: Test class contains implicit setup, but test case in class is completely independent of that setup.
- *Dead Fields:* Test class or its super classes have fields that are never used by any method in the test class.
- *Lack of Cohesion of Test Methods*: Test methods grouped into one class are not cohesive.
- *Obscure-In-Line-Setup:* Covers too much setup functionality in the test class.
- *Vague Header Setup:* Fields are initialized in the header of the test class but never used in the implicit setup.

Fixture usage is analyzed by using static code metrics. After invention of code metrics they mapped them to fixture strategies. For some, certain thresholds had to be set. The analysis was done in 3 steps; Fact extraction (metrics), analysis (using metrics, mapping to fixture strategies) and presentation. Greiler et al. found that fixture related test smells occur in practice, where 5 to 30 % of the methods / classes are involved. Developers recognize the importance of high test code quality and that reduction in test code quality can lead to maintenance problems. They sometimes introduce fixture smells on purpose.

Greiler et al. automated the detection process of Fixture Smells. Their process can be extended to automatically detect test bugs. A Fixture Smell is also a symptom of test bugs in the *Resource Handling* category in the paper by Vahabzadeh et al. and can be used to find *Resource Handling* bugs in test code.

2.5 Is This a Bug or an Obsolete Test?

When regression tests fail, it can signal a bug in production code or an obsolete test. Automatic techniques exist to find the location of these bugs and solve the bugs. However, the programmer needs to know if the defect is in production or test code.

A machine-learning approach can help. The Hao et al. adopted the Best-First Decision Tree to classify regression test failures as coming from production or test code [6]. Seven features of tests (static and light-weight dynamic) are identified to train a classifier.

The study is performed on two medium size Java systems. Three studies were conducted: training set in same version as test set, in different versions, or in different programs. For the same-version experiment the machine learning classification method produced 80% accuracy. For different versions the accuracy was even higher, around 90%. Different programs resulted in decreased accuracy (around 60%). 50% is considered guessing.

The term *obsolete test* might be misleading, since Hao et al. only determined if a test failure was due to a failure in production code, or a failure in test code. When the failure is in test code, the developer can still repair the test to work with the (new) production code. The high accuracy obtained by Hao et al. is promising for building a detector for test bugs. However, training data is needed, and their approach deteriorates when the training set comes from a different program. It would be interesting to see what happens when the training set comes from multiple different programs which is exactly what Vahabzadeh et al. produced .

2.6 A Bayesian Approach for the Detection of Code and Design Smells

Current research on smell detection only focused on Boolean classification. Khomh et al. implemented a detection method using Bayesian Belief Networks (BBN)[9]. A BBN assigns probabilities to classes having a code smell. This allows analysts and developers to prioritize issues, which is not possible using the Boolean approach of classification.

Khomh et al. trained a BBN with data from two medium sized open source Java projects. They illustrated their approach on the Blob (God Class) anti-pattern. First they instructed undergraduate and graduate students to identify all Blob classes in the Java programs under study. They constructed a BBN using rules established by previous research (Moha et al. [12]). The BBN was then trained by using information about the number of identified Blob classes in both programs.

Khomh et al. showed that a BBN can be used to prioritize design issues by giving probability to classes potentially having a design smell. Their method was evaluated against the DECOR method from Moha et al.[12] The BBN approach showed better results (accuracy and recall). Accuracy was around 60 %, where recall of 100 % was achieved. The BBN can also be calibrated for a specific context (e.g. adjusting the impact of rules) to increase its accuracy even further.

The test bugs categorized by Vahabzadeh et al. [14] can be used as a training set to train a BBN to find test bugs. The process will then be automated, and validating their classification scheme will require less manual inspection. In a later stage the BBN network can be extended to classify issue reports before we know it is a test code failure. It will then serve as a tool for developers to help them in prioritizing issue reports.

2.7 Concept-Based Failure Clustering

DiGiuseppe et al. [4] proposed a novel way to cluster program failures based on semantics. The stack trace responsible for the failure is compared based on the occurrence of words, e.g. in variables and comments. This paper is a proof of concept, and shows for a medium sized system that semantic based clustering produces less clusters as control-flow based clustering while maintaining the same accuracy.

Issue reports contain a lot of semantic information. This paper showed that it can be promising to incorporate this information into a classification model for test bugs, based on clustering.

2.8 Empirically Detecting False Test Alarms Using Association Rules

Herzig et al. [7] tried to reduce the number of false alarms in test bugs during integration testing. They used associate learning to classify issues about test bugs being false positives. Accuracy was about 85-90% with the number of correctly detected false positives at the end ranged from 34 to 48%. They mainly investigated integration tests on two Windows systems for a period between one and two years.

Integration tests consist of a number of test steps. When one test step fails the integration test fails. The integration test is not immediately aborted, but continues to run resulting in an array of failed or passed test steps. These arrays where used as antecedents (left hand side of association rule) and linked to either a false positive or an actual production code bug. If a rule occurs enough, is non-contradictory, and has high confidence (the rule predicts a given outcome with high accuracy), the rule is kept.

For *Windows 8.1* the maximum net gain (in man hours) was 1.7 hours per day. For *Dynamics XS* it was 14 minutes per day. In a real world company one still needs to investigate each case, since there exist cases where a bug is introduced into the system but classified as False Positive.

2.9 Detection Strategies: Metrics-Based Rules for Detecting Design Flaws

Marinescu [11] proposed a way to systematically detect design flaws using a number and composition of metric-based rules. The purpose of his work was to detect the disease instead of symptoms (metrics), which could potentially point to a software disease, but require an analyst to draw conclusions.

The detection strategy consists roughly of two steps. Step 1 is data filtering, where the detection strategy defines whether to use relative, absolute, statistical or interval filtering. Step 2 is a composition of data filters using Boolean logic and set operators. This results in a set of candidates potentially affected by the design flaw.

Marinescu used his method to design detection strategies for 9 well-known design flaws and validated the approach on one open source system with two different versions. Average accuracy was 67%, where 50% defines mere chance.

Marinescu framework can be extended to also detect test bugs. The challenge lies in choosing the right metrics, and correct thresholds. There exists literature about some test bugs [5, 6, 7, 10], but the field is fairly new.

2.10 An Empirical Analysis of Flaky Tests

Flaky tests are tests that sometimes fail due to concurrency issues [10]. Luo et al. started from commit messages and sampled 201 commit messages (previously identified as fixing a flaky test). For each of these commits they determined the root cause of the flaky test, how it manifests itself in the code, and how it was fixed. The top three categories of flaky tests are:

1. `async wait` (45%): A test makes an asynchronous call and does not wait properly for the resources to become available. Almost all flaky tests (96%) fail independent of the platform. Roughly a third uses a simple method call with time delays to enforce ordering. 85% involve only one ordering and do not wait for external resources. This means that flaky tests in this category could be detected by inserting one time delay in a certain part of the code without the need to control the external environment. Using `waitFor` (often instead of `sleep`) causes 54% of bugs to be fixed.

2. `concurrency` (20%): caused by different threads interacting in a non-desirable way, but not in the `asynch wait` category. Almost all bugs in this category involve two threads or can be reduced to two threads. 97% fails due to concurrent accesses on memory objects only. Concurrency bugs are solved in a number of ways; adding locks, making the program deterministic or changing conditions.

3. `test order dependency` (12%): The outcome of a test depends on the order in which tests are executed. The problem may arise when two tests depend on a common resource, which is not properly managed. About half (47%) are caused by a dependency on external resources. By cleaning the shared state between test runs proved successful in solving the bug in 74% of the cases.

Other causes of flaky tests are fixed in a variety of ways. When a fix is made in the code under test (CUT), most fixes (96%) also fix a bug in the CUT.

Luo et al. provide patterns for detecting flaky tests, which can be incorporated into a test bug detection tool. The patterns are based on static source code analysis.

2.11 Are Test Smells Really Harmful? An Empirical Study

Bavota et al. [3] talk about the diffusion of test smells among test classes. There is an high percentage of test classes having at least one test smell. Test smells are categorized by hand, based on the list of van Deursen. The correlation between test smells and several metrics *(LOC, #testclasses, #testmethods, #tLOC, ProjectAge, testCodeAge, teamSize)* reveals importance of test smells. Some

smells are more prevalent as others. Assertion roulette, which occurs when a method has multiple assertions and at least one without explanation , is the most occurring (55%). Eager test, which uses more than one method from the tested class comes second (34%).

They also studied the effect of test smells on maintenance. They held a questionnaire among different groups of students and software engineers. Among groups, experience level was roughly the same. Between groups, experience level varied. Bavota et al. measured correctness in answering questions about tests with and without test smells. For each question, the time to answer that question was measured as well.

The eager test smell has a strong significant negative correlation with precision. Test subjects were more precise in answering questions about the refactored test method as opposed to the method still containing the eager test smell. The eager test smell is the only smell that showed a significant strong correlation between time. Test subjects took more time when answering the question with the test smell as opposed to without.

Their main conclusions were that test smells generally have a negative impact on correctness. No significant impact on time was found. Experienced developers experience negative impact on correctness with only 3 test smells, as opposed to first year bachelor students. This group experienced difficulty with at least 6 test smells.

Initially, I wanted to replicate the study of Vahabzadeh et al. in categorizing test bugs. The authors spend around 400 hours categorizing 443 issue reports, which is a little under an hour for one issue report. The authors were more experienced software engineers than I am, so this paper confirmed to me that a manual replication of their test bug categorization process was infeasible, even on a smaller dataset. It is also interesting to investigate the relation between test smells and test bugs. Test smells can sometimes be the symptoms of test bugs.

2.12 Finding Bugs is Easy

There exist many tools that find bugs with extensive analysis and complicated algorithms. Hovemeyer et al. found some cases were production code contained trivial bugs, easily detectable by a bug pattern. On this bases they developed FindBugs, a static analysis tool for Java Bytecode [8].

FindBugs works on the basis of a bug pattern. A bug pattern is a part of source code that usually contains a bug. They illustrated several patterns, their rationale, and showed evidence of the occurrence of these patterns in real world software systems. FindBugs is designed to be easily extended by plug-ins.

The aim of FindBugs is not to be as accurate as possible, or have a high recall. It is designed to be easily incorporated into the development process, and find relatively simple bugs. Hovemeyer et al. showed, for a number of bug patterns, that FindBugs has an acceptable false positive rate.

Vahabzadeh et al. listed a number of test bug categories, which can be translated into FindBugs bug patterns.

2.13 Experiences Using Static Analysis to Find Bugs

FindBugs, a popular static analysis tool for Java Bytecode is analyzed regarding its usage in real world settings. Ayewah et al. explain their experience with developing and deploying FindBugs [2]. FindBugs was introduced in 2004 [8].

Ayewah et al. deployed their tool at Google. They found that FindBugs usage is dependent on formal policies. Without, it is up to the developer how FindBugs is used, and usage may very wildly.

They also found that developers review a high percentage (around 90 %) of high risk warnings. Lower level warnings are reviewed as well, but are context-specific. Developers fix warnings ad-hoc, can use annotations to suppress warnings, or use filtering techniques to filter out uninteresting warnings or false positives. There are a number of strategies to keep track of FindBugs warnings, and keep them relevant. Companies can use store warnings in a database, collect and send warnings after each day via email, immediately fix warnings or incorporate warnings in their review policies.

Ayewah et al. also discuss the limitations of FindBugs, which is useful when considering building new detectors, i.e. for test bugs.

2.14 A Comparison of Bug Finding Tools for Java

Rutar et al. studied several bug finding tools for Java, and compared their results [13]. FindBugs is a static analysis bug finder that processes Java Bytecode and is based on syntax and dataflow analysis. JLint is similar to FindBugs, but *"also includes an inter-procedural, inter-file component to find deadlocks by building a lock graph and ensuring that there are never any cycles int he graph"*. FindBugs' dataflow analysis is restricted to the method level. PMD performs syntactic checks on source code, and is more focused on detecting violations in stylistic conventions. Bandera is a verification tool based on model checking. The programmer can use annotations to include specifications to be checked. Without annotations, Bandera primarily checks for standard synchronization properties. ESC/Java is based on theorem proving and performs formal verification of properties of Java source code. The programmer adds preconditions, postconditions and loop invariants in the form of annotations. Without annotations ESC/Java looks mainly for null pointer dereferences and array out-of-bounds errors.

Interestingly, Rutar et al. found no significant correlation among pairs of tools. They tested for a relation between code size (lines of code with spaces and comments removed) and number of reported warnings on the class level. They also tested if the number of reported bugs for one tool correlated with other tools.

Rutar et al. proposed a meta-tool that combines the results of different tools and ranks classes based on two metrics; normalized warning total and unique warning total. For the latter, they counted the same error messages as one, e.g. cascading error messages for JLint. For the former they counted all error messages a tool issues per class, added the result for all tools, and normalized the result by the maximum value of issued errors for that class. They found a significant correlation between the two metrics. When a class scored high in normalized warning total, it could score very low in unique warning total, and vice versa. The goal of these metrics was to give developers information on potentially interesting classes that individual tools would miss. Rutar et al. found several examples where their meta-tool indicated a class to be important, but individual tools would have rated it much less important.

Bibliography

[1] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *Software Engineering, IEEE Transactions on*, 40(11):1100–1125, 2014.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, 2008.

[3] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[4] N. DiGiuseppe and J. A. Jones. Concept-based failure clustering. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, page 29. ACM, 2012.

[5] M. Greiler, A. van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 322–331. IEEE, 2013.

[6] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang. Is this a bug or an obsolete test? In *ECOOP 2013–Object-Oriented Programming*, pages 602–628. Springer, 2013.

[7] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *Companion Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.

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

[9] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 305–314. IEEE, 2009.

[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[11] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[12] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3-4):345–361, 2010.

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

[14] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 101–110. IEEE, 2015.

[15] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. *Refactoring test code*. CWI, 2001.

[16] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.