Continuous Mutation Testing in Modern Software Development

Master's Thesis

Ioana Leontiuc

Continuous Mutation Testing in Modern Software Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ioana Leontiuc
born in Timişoara, România

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Continuous Mutation Testing in Modern Software Development

Author: Ioana Leontiuc
Student id: 4515218
Email: I.Leontiuc@student.tudelft.nl

Abstract

Modern software is being built in a continuously integrated fashion, in order to overcome the challenges that come with developing large software systems from many contributors. The cornerstone of continuous integration is the testing step, since it is supposed to protect the system from changes that might disrupt correct behavior. Mutation testing is a method that checks the fault finding capability of a test suite. Current CI settings do not implement a step that checks how thorough the test suite is.

Therefore, the goal of this thesis has been to explore how mutation testing can be applied to changes under analysis in a continuous integration setting. Since there is no infrastructure to support this, in order to conduct our study we developed OPi+, a prototype tool for experimenting the infrastructure required for a continuous mutation testing approach. Using real-world systems for analysis, we give initial evidence of the continuous mutation testing usefulness in terms of costs and benefits when applied to realistic software changes. The empirical study is based on analysis performed on the entire commit history of the popular open source Java Maven systems.

Through our study we defined 5 types of outcomes together with a continuous mutation testing behavior flow and additional analysis that streamlines current mutation testing practices. We showed not only that mutation testing in a CI environment requires significantly fewer resources but they are also within the limits required by a CI pipeline. Through our study we also identify unmutable code for which we propose appropriate unimplemented operator set. We also study the evolution of surviving mutants with regards to their impact on the systems‘ technical debt.

In our study, we showed initial evidence that mutation testing can successfully be made compatible with a CI environment. We therefore propose a few ideas that could possibly further streamline continuous mutation testing.

Thesis Committee:

Chair/Supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. Claudia Hauff, Faculty EEMCS, TU Delft
Committee Member:	Dr. Cynthia Liem, Faculty EEMCS, TU Delft

Preface

"No man is an island, entire of itself..." [28]

I am very content to be able to express my sincere gratitude to my supervisor, Prof. Dr. Arie van Deursen for his excellent advice and patient guidance. The current work would not have been completed without his extensive efforts, extraordinary ability to motivate me and genuine interest in my work.

I am also thankful to my extremely amiable teachers and colleagues at SERG. I would like to show my appreciation to Mozhan Soltani and Qianqian Zhu for their availability, their help and suggestions.

My sister has always encouraged me to study and actively supported all my academic endeavours. Thank you, I would not be here, physically and metaphorically, without your care and affection. I am grateful to all my friends, who have inspired and cheered me up. Special thanks to Kyriakos Fragkeskos for his efforts and time spent in order to save my time.

Ioana Leontiuc
Delft, the Netherlands
August 15, 2017

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Problem Formulation . . . 2
- 1.2 Proposed Approach . . . 3
- 1.3 Empirical Evaluation . . . 3
- 1.4 Outline . . . 4

2 Background 5
- 2.1 Continuous Integration . . . 5
- 2.2 Version Control . . . 6
- 2.3 Test adequacy . . . 7
- 2.4 Code review . . . 7
- 2.5 Operias . . . 8
- 2.6 Mutation testing . . . 8

3 Proposed Solution 15
- 3.1 Process flow . . . 15
- 3.2 Implementation . . . 17

4 Empirical Study 27
- 4.1 Research Questions . . . 27
- 4.2 Project Selection . . . 28
- 4.3 Experimental Procedure . . . 29
- 4.4 In Depth Analysis of Output Types . . . 32

5 Empirical Results 37

5.1 RQ1: How is the mutant generation, time and developer effort of mutation testing impacted in a continuous integration environment? 37
5.2 RQ2: What are the implications of continuous mutation testing on the completeness of currently available operator sets? 45
5.3 RQ3: What feedback information do the surviving mutants offer to developers in the context of a continuous integration setting? 47
5.4 RQ4: Is there a correlation between surviving mutants and system characteristics such as: code coverage, churn and bad smells? 49
5.5 RQ5: Can continuous mutation testing be used in the code review process? 55

6 Discussion **59**
6.1 Result interpretation 59
6.2 Threats to Validity 63
6.3 Future Work 64

7 Related Work **69**
7.1 Improving Continuous Integration 69
7.2 Techniques to Optimize Mutation Testing Usability 70

8 Conclusions **73**

Bibliography **77**

A Manual Analysis **83**

B Missing Test Case Contributions **91**
B.1 JSoup 91
B.2 Commons Compress 93
B.3 Commons IO 94

C Selected Object-Oriented Mutation Operators from Literature **95**
C.1 Encapsulation 95
C.2 Polymorphism 95
C.3 Inheritance 96
C.4 Java-Specific Features Some 96

List of Figures

2.1 Changes moving through the deployment pipeline [39] 6
2.2 Mutation testing operators that can be applied on the highlighted area [58] . . . 11

3.1 Continuous Mutation Testing Behavior Flow 18
3.2 OPi+ Architecture 19
3.3 UML Representation of the Main Data Structures in OPi+ 20
3.4 OPi+ Architecture 3.2 with External Dependencies 25
3.5 Continuous Mutation Testing Decision Flow in OPi+ 26

4.1 OPi+ Evaluation Data Gathering Process
Each text line represents a column in the data table.
Each box represents the step that generates the data for the specific columns
. 31
4.2 Data Extraction point in OPi+ Architecture 31
4.3 Label 5 in depth analysis 34

5.1 Jsoup File Size vs Pitest and OPi+ analyzed lines
(p=0.0001592; p=0.540378) 41
5.2 Common Compress File Size vs Pitest and OPi+ analyzed lines
(p=0.0002855; p=0.569648) 42
5.3 Commons IO File Size vs Pitest and OPi+ analyzed lines
(p=0.0036486; p=0.014331) 43
5.4 JSoup: Surviving mutants generated by OPi+ from the entire change history vs survived mutants generated by Pitest for the last version of the same file
(p=0.848864) 44
5.5 Commons Compress: Surviving mutants generated by OPi+ from the entire change history vs survived mutants generated by Pitest for the last version of the same file (p=0.252013) 44
5.6 Commons IO: Surviving mutants generated by OPi+ from the entire change history vs survived mutants generated by Pitest for the last version of the same file ($p < 0.0001$) 45

5.7 Surviving Mutants in JSoup Commit History 46
5.8 Surviving Mutants in Commons Compress Commit History 46
5.9 Surviving Mutants in Commons IO Commit History 47
5.10 Distribution of Type-5 lines in JSoup 47
5.11 Distribution of Type-5 lines in Commons Compress 48
5.12 Distribution of Type-5 lines in Commons IO 48
5.13 Jsoup Branch Coverage for each Class vs Surviving Mutants found by OPi+ . . 50
5.14 Commons Compress Branch Coverage for each Class vs Surviving Mutants found by OPi+ 51
5.15 Commons IO Branch Coverage for each Class vs Surviving Mutants found by OPi+ 52
5.16 JSoup Churn vs Survived Mutants computed with OPi+ for each class (p=0.0029194), highlighted with orange Bad Smell Classes 53
5.17 Commons Compress Churn vs Survived Mutants computed with OPi+ for each class(p=0.160202), highlighted with orange Bad Smell Classes 53
5.18 Commons IO Churn vs Survived Mutants computed with OPi+ for each class($p<0.0001$) 54

7.1 Distribution of common build error categories [63] 69

Chapter 1

Introduction

"We learn by putting stuff into production. We can ask people till they are blue in the face "What do you want?", but it's only when you actually give people something, they will tell you what they don't want". This is what Martin Fowler calls "the inevitable feature of software development" [34]. The users of software products usually clarify the requirements while they see intermediate deliverables. This dynamic is the main reason why modern software development is a continuous process. Continuous software development is not hindered by changing requirements, but instead profits from them by transforming them into continuous improvement feedback. This results in constantly running software that keeps up with the volatile requirements of the client.

The development practice that allows multiple developers to integrate changes on the same code into a shared repository is called continuous integration [39]. Every change a developer makes must pass a given test suite before being integrated in the master version of the system. The master version of the system should always be in a correct state since it functions as the final product at any given time before the completion stage. Therefore the success of the integration process depends on the test suite capability to catch faulty behavior that might break the system into production.

Once the changes have been continuously integrated, the new version of the system is automatically being processed into a deployable product. Continuous delivery is a software development discipline that allows software features which have already been implemented to be released to production at any time [39]. Even though the change could be made live in production from a technical point of view, this is ultimately a management decision. Continuous delivery means that we are at the point where all changes are being deployed continuously, but are delivered to the clients based on management flags [34]. The advantages of continuous delivery are the following: smaller deployment risks, providing a sense of real progress and making use of user feedback [34]. A prerequisite of continuous delivery is developing code in a continuous integration environment.

A successful continuous integration process is restricted by the test suite behavior, meaning that the behavior of the test suite decides whether a change will be integrated in the system. These restrictions may be: selecting a successful subset of the test suite as a prerequisite, selecting a specific threshold of passed tests or including a quality metric test suite. The quality of the test suite is based on different aspects of the system under test,

such as code coverage or test code quality.

Code coverage is most often used as a test adequacy metric in continuous integration processes [65]. Although it may be computed in different ways, the most important aspect is whether a piece of code is being executed by at least one test case or not. It does not analyze how that code is being tested, so it is not a comprehensive indicator metric for test quality since it cannot detect faults. Code coverage can only detect codes that are never executed by any test case, but does not analyze the way that codes are being executed. Consequently, it is possible to have test covered faulty code.

A method that can measure the fault detection capability of a test suite in the code base is mutation testing. This approach is a step further than code coverage because it may be used to check whether existing tests are actually able to detect certain types of faults injected in the code covered by them. By detecting the areas in the code that are not thoroughly tested, mutation testing automatically provides improvement feedback for the test suite by pointing to insufficiently tested code.

1.1 Problem Formulation

The continuous integration process requires an additional stage to assess the adequacy of a test suite in order to have a more reliable delivery to production. A widely studied solution for assessing the fault detection ability of a test suite is mutation testing [41]. Since the initial proposal [46] in 1971, many improvements have been implemented, the approach has been tested in empirical studies and applied in many languages and different settings [41].

Unfortunately, the current practices of applying mutation testing are not compatible with continuous integration processes. The incompatibility is due to several factors:

1. very expensive method which requires a significant amount of processing power and time [41]
2. It is currently being applied on the entire system, while continuous integration deals with the change and does not analyze the entire system each time
3. requires a complicated and lengthy manual post analysis due to the unfiltered output of mutation testing
4. current techniques for mutation testing cost reduction are not tailored for CI environments, but for stand-alone laboratory analysis which is not based on many small changes.

The goal of this thesis is to *explore how mutation testing can be applied to changes under analysis in a continuous integration setting*. We particularly investigate the following:

1. How can we set up an infrastructure that allows for an efficient analysis of changes that are processed in a continuous integration server?
2. What are the costs and benefits of such an infrastructure when applied to realistic software changes?

Continuous mutation testing implies that the continuous integration process would include a specific step for checking the effectiveness of the test suite. The mutation process would be applied only to the changed code. This hybrid solution cancels the deficiencies of both methods. Not only does mutation testing become manageable from a resource perspective, but also the CI environment is able to integrate code with a proven degree of confidence. An overview of this hybrid approach is given in the next section.

1.2 Proposed Approach

Continuous mutation testing is a concept independent of the specific implementation of the mutation testing tools. Although mutation testing is being used to some extent in real world projects, it was never integrated in a CI environment. Therefore, there is no infrastructure to support continuous mutation testing. The purpose of this study is to analyze whether mutation testing can improve the development process by integrating it in a CI environment. In order to answer this question we developed OPi+, a prototype tool to conduct experiments with change-driven mutation testing. In this environment, we adapted current technologies to meet the requirements for continuous mutation testing by customizing the core of the tool or extending the features with a custom proxy. The way we integrate continuous mutation testing with the flow of modern software development mimics a related project Operias [59]. The Operias system is a tool meant to help the process of code reviewing based on the suggestions on the progress of code coverage. A customized version of this system is the first step in the continuous mutation testing process.

1.3 Empirical Evaluation

We use OPi+ to automate all the steps of a system‘s development process in chronological order, also inserting mutation testing in each iteration of the process. This way, we mimic a continuous mutation testing approach in a real development process. We record all commits that should be further analyzed, all commits that were ignored and compute the impact of each commit within the context of change-based mutation testing.

After we have obtained an overview of all the processed code through the continuous mutation testing method, we conduct an empirical study to analyze how the follow up actions suggested by OPi+ can help improve the test suite. By doing this we can assess to what extent this approach can improve the continuous delivery of a well tested system and the usability in terms of:

1. processing power
2. time
3. developer effort

required in relation to the usefulness of the feedback computed by the approach.

The analyzed data from the experimental environment of continuous mutation testing has implications in how the current method can be improved. These implications relate to the reduced costs of continuous mutation testing, the completeness of the currently available mutation operator set and the potential impact of surviving mutants in improving the system. All these implications make a case for setting up the infrastructure for this hybrid solution which provides continuous feedback for specific missing test cases. The solution in question also helps highlight critical code areas that need testing and contributes to creating a more trustworthy build process.

1.4 Outline

This thesis presents the study on continuous mutation testing. Chapter 2 presents a background on the traditional mutation testing approach and current continuous integration practices. Chapter 3 describes the conceptual proposed solution for continuous mutation testing and its implications. Chapter 4 presents the design of the empirical study that assesses the effects of the approach. We present the results of the empirical study in Chapter 5. Based on the results, we attempt to answer five research questions related to our main objective, "can mutation testing be compatible with a continuous development environment?". 6 presents an in-depth analysis of the empirical results and their implications, threats to validity and proposed future work. Chapter 7 describes the related studies in the area of making the continuous integration process more effective and making mutation testing more usable. Chapter 8 presents the conclusions drawn from the outcome of this hybrid approach, based on the empirical study.

Chapter 2

Background

2.1 Continuous Integration

"Its hard enough for software developers to write code that works on their machine. But even when that's done, there's a long journey from there to software thats producing value - since software only produces value when its in production." Martin Fowler [32].

Continuous Deployment is a development process that takes the raw code and transforms it into a client deliverable product after every change within the system [39]. All steps that take the change test it and then integrate it in the master version of the system form the *continuous integration pipeline*. Continuous Integration, a sub part of the Continuous Deployment process, is an automated software delivery process.

The Continuous Integration process is triggered by a change to the code base. A more detailed cause-effect flow within the build pipeline is described in Figure 2.1. Even though these steps are custom to every project, they can be classified in a few chronological categories as described by Humble et al. [39]:

- The commit stage - asserts that the system works at the technical level (compile, pass test suite, run code analysis)
- Automated acceptance test stages - assert that the system works at the functional and nonfunctional level (meets user requirements)
- Manual test stages - can include exploratory testing environments, integration environments, and UAT (user acceptance testing)
- Release stage - delivers the system to users, either as packaged software or by deploying it into a production or staging environment (a staging environment is a testing environment identical to the production environment).

The CI pipeline makes releasing software easy if the automated tests are up-to-date and do not require a lot of manual compensation. "Incorporating testing into every part of your delivery process is vital to getting work done. Since our approach to testing defines our understanding of done, the results of testing are the cornerstone of project planning.

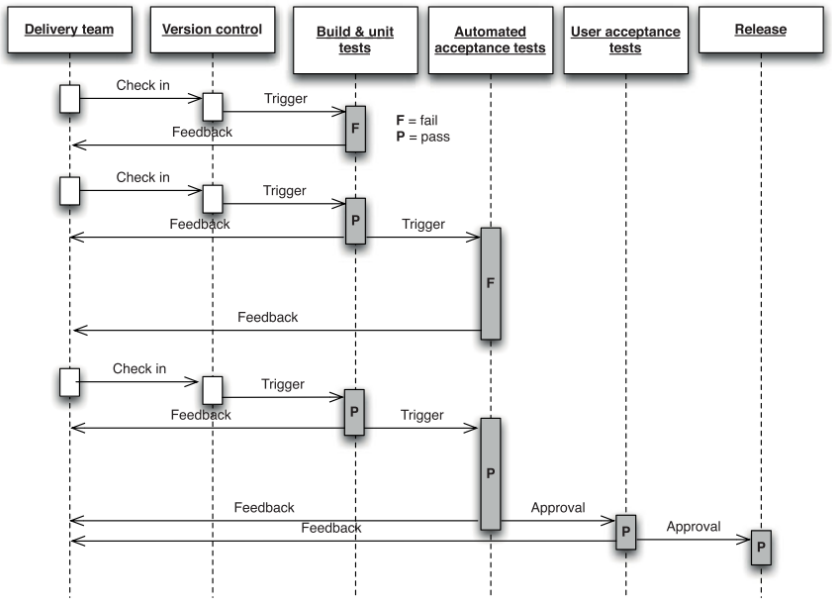


Figure 2.1: Changes moving through the deployment pipeline [39]

Testing is fundamentally interconnected with your definition of "done, and your testing strategy should be focused on being able to deliver that understanding feature by feature and ensuring that testing is pervasive throughout your process" [39]. The underlying idea behind Continuous Integration is that if the code has a comprehensive test suite that passes in a production-like environment then it is ready for release. This accentuates the need for high quality tests.

"A smooth path to the release" is highly dependent upon a high quality test suite. In order to assess the quality of a test suite, development teams use code coverage [36]. Unfortunately high test coverage does not necessary imply a good test suite [39] [40]. Too often manual testing is left to compensate for a poor automated test suite [39]. Since humans are not good with repetitive, mundane yet complex tasks this process may lead to poor quality software [39].

The CI process can become more effective if an additional form of assessing the quality of a test suite is adopted: mutation testing, as we will discuss in the next section.

2.2 Version Control

An effective way to prevent software aging is by considering software changes to be the center of the development process [30]. History tracking, currently relies on Version Control Systems (VCS). These systems treat the software system as a collection of text files and log all textual changes. However this log approach cannot convey the impact a change has on the system. Nevertheless a proper VCS is the cornerstone of continuous integration. Our prototype experiment tool is using Git as the version control system since the empirical

study is based on open source projects hosted by Git. This is a very popular platform with over 61 million projects hosted [9].

Pairing VCS analysis with mutation testing has the potential of canceling one of the biggest challenges mutation testing faces: manual analysis of surviving mutants. By timing mutation testing with the commit time, the mutation process may become maintainable by focusing on the new few changes. If we match mutant operators with types of change analyzed in difference analysis we can prioritize the mutations. Furthermore, the programmer responsible for the change can analyze the related mutants, in order to identify the mutants that need manual analysis.

2.3 Test adequacy

White-box testing consists of test cases that are build based on the internal structure of the program, such as instructions and branches [25, 18]. Some tests quality criteria are line and branch coverage.

Line coverage or statement coverage is the basis for the simplest and weakest form of testing [38]. This type of coverage is computed by counting the number of instructions that are being executed by at least one test. It is considered that the higher the coverage number, the better the test suite is. Nevertheless, achieving a full line coverage is not possible for unreachable code. Whether a line is unreachable or not cannot be automatically detected, since it is an undecidable problem [68].

Branch coverage is computed based on the decision paths executed by the test cases. This is a stronger requirement for a more adequate test suite [38]. If all branches are tested then also all statements will be tested. Nevertheless, this criteria is not very thorough when there are more conditions. The number of inputs for more combinations increases exponentially.

Cobertura [4] is an open source project that computes line and branch coverage of a Java system. It is based on JUnit test suites. More than this, Cobertura can also compute the McGabe complexity of each tested method. The McGabe metric [50] measures the complexity of a method and it is computed on the number of linearly independent paths through the analyzed piece of code. It is also compatible with Ant or Maven projects and it generates results in the form of XML reports.

2.4 Code review

Code reviewing is a common software engineering practice meant to find defects [20]. This is done by having a developer approve the changes made by another developer. This step is a part of the software integration process. This approach has added benefits such as knowledge transfer, increased team awareness and creation of alternative solutions to problems as found by Bacchelli et al. [20] empirical study. Developers that took part in the study practiced code review as means to find defects, improve the code, avoid build breaks, improving the development process and other social team related aspects. According to this study, these actions are necessary since the current tools are not yet able to replace the manual

analysis in all cases. It also showed that there is a need for a more thorough reviewing and understanding the changes within the continuous integration process.

2.5 Operias

In order to implement a continuous mutation testing experimental environment we need a step that analyzes the change submitted in one commit. In order to improve the performance of OPi+ we implement a pre-filtering feature that triggers the mutation process provided by Pitest. For this step we decided to make use of Operias, a review tool for developers based on change in code and change in test coverage [59]. Operias generates HTML and XML reports based on any two versions of a project and its test suite. The report contains information about the change in code and in test coverage computed via Cobertura [4], a branch coverage tool for Java.

Operias can analyze Maven projects written in Java. The mutation testing tool we use, Pitest, has the exact same limitation on the project scope, java project with Maven organization. Current Operias implementation also provides feedback for the reviewer on GitHub, having implemented a proper communication channel with GitHub. We reuse part of this communication channel for OPi+.

The main usage scenario for Operias is code reviewing a pull request, which is a set of several changes that build up to the same goal, like a new feature. Even though OPi+ is tailored for a commit approach the algorithm is the same. The two versions Operias will analyze are subsequent versions of the system. Another advantage of using Operias is that we can also analyze the benefits of mutation testing as part of the review process.

2.6 Mutation testing

2.6.1 Process overview and terminology

Mutation testing is a fault-based, white box testing technique, which seeds faults in the original system by making small syntactical changes in order to test the fault detection capability of a test suite. The Competent Programmer Hypothesis [46] states that the programs produced by programmers are almost correct thus the faults within the system can be corrected with few syntactical changes. At the same time, the Coupling Effect [46] states that the fault which requires complex changes can be reduced to several simple errors. This means that if a test suite can detect small syntactical faults, the complex faults are also detected [55]. Based on this, the mutation testing approach can asses the effectiveness of a test suite in terms of its ability to detect faults. [41].

The first step in the mutation testing process is making a change in the original system. This change is meant to break the system. It could be as simple as swapping an operator like $<$ to $>$. These changes are meant to break the original system, mutate it, thus generating a faulty version of the system called a *mutant*. A useful mutant should behave abnormally in the scope of the system‘s requirements [41].

The transformation rule that generates a new mutant is called a *mutation operator* [41]. An operator can be any change pattern based on the syntax of a software language. They can

be specific for each language making them *language specific operators*. Mutation operators have been applied in languages with object-oriented-features such as Java, C#, Eiffel, C++, JavaScript, and Delphi [62]. Once an operator is applied to the original system, a faulty version is being generated. This mutated version of the original system is called a *simple mutant* or *first order mutant*. In contrast, a *complex mutant* is a faulty version of the system that is generated by applying several operators. We call this an *n-th order mutant*, where n is the number of applied changes. Based on the Competent Programmer Hypothesis [46], and the Coupling Effect [46], it is believed that the changes performed by the operators are similar to real-life issues.

The mutants generated from the original system are all run against the test suite. Ideally at least one test should fail thus showing the new version is indeed faulty. If at least one test fails the mutant is considered *killed* [41]. However it could be that the mutated version passes all tests just like the original version, in which case it is referred to as a *surviving mutant* [41].

Mutation testing can also provide concrete overview of the test suite quality through the mutation score. Once we know the status of each mutant, survived or killed, the method provides a *mutation score* that grades the quality of the input test set [41]. We must note that an *equivalent mutant* is a survived mutant that generated an identical version of the system. Once equivalent mutants are filtered out of the surviving mutants we are left with valuable cases that point to specific missing test cases. The mutants that are not killed by the test suite and are not equivalent to the original system are the ones that provide useful feedback. This score is computed as:

$$\frac{KilledMutants}{Allmutants - EquivalentMutants} = \frac{DetectedFaults}{SeededFaults}$$

.

We apply the mutation testing process to the example in Listing 2.1.

```
1  public class BankAccount {
2
3    private double balance;
4
5    public boolean deposit(double amount) {
6      if (amount > 0.00) {
7        balance = balance + amount;
8        return true;
9      }else{
10       System.out.println("amount cannot be
       negative");
11       return false;
12     }
13   }
14
15
16   public boolean withdraw(double amount) {
17     if (amount <= 0){
18       System.out.println("amount cannot be
       negative");
19       return false;
```

```
20      }
21      if (balance < 0){
22          System.out.println("account is
        overdraft");
23          return false;
24      }
25      balance = balance - amount;
26      return true;
27    }
28
29  }
```

Listing 2.1: Literature Survey root example

We have full branch coverage for deposit method by writing the tests in Listing 2.2

```
1   @Test
2   public void depositIfCoverage() {
3     BankAccount acc = new BankAccount();
4     assertTrue(acc.deposit(10.00));
5   }
6
7   @Test
8   public void depositElseCoverage() {
9     BankAccount acc = new BankAccount();
10    boolean result = acc.deposit(-10.00);
11    assertEquals(result, false);
12  }
```

Listing 2.2: Full branch coverage for deposit()

If we mutate line 7 from our example Listing 2.1 using an operator that changes + to − our mutant passes the full coverage test suite. This means that our second version of the system that deposits money by erroneously deducting the sum (balance = balance - amount;) is a surviving mutant. The mutation score is currently $0 = 0/1 - 0$.

A high quality test suite should catch the faulty behavior with at least one test case. We kill the surviving mutant by adding the test described in Listing 2.3. After improving our test suite the mutation score increases to $1 = 1/1 - 0$.

```
1 @Test
2   public void afterMutation() {
3     BankAccount acc = new BankAccount();
4     acc.deposit(10.00);
5     boolean result = acc.withdraw(10.00);
6     assertEquals(result, true);
7   }
```

Listing 2.3: Test to kill faulty deposit surviving mutant

Traditional Mutation Testing requires all possible mutants to be generated and run against the entire test suite [41]. Given a specific set of operators, we can generate mutants by changing all areas marked in Figure 2.2. The yellow highlighted areas indicate basic non object oriented operators such as changing a conditional boundary, a constant value or changing the return value. All these operators are very common and have various names in the literature [41].

```java
public class BankAccount {

    private double balance;

    public boolean deposit(double amount) {
        if (amount > 0.00) {
            balance = balance + amount;
            return true;
        }else{
            System.out.println("amount cannot be negative");
            return false;
        }
    }


    public boolean withdraw(double amount) {
        if (amount <= 0){
            System.out.println("amount cannot be negative");
            return false;
        }
        if (balance < 0){
            System.out.println("account is overdraft");
            return false;
        }
        balance = balance - amount;
        return true;
    }

}
```

Figure 2.2: Mutation testing operators that can be applied on the highlighted area [58]

2.6.2 Limitations

The main disadvantage of the mutation testing process is that it requires a significant amount of **computational resources**. This is due to the fact that all mutated versions of the system have to be compiled and run against the test suite.

To remedy this problem, efforts have been made to reduce this cost. The cost reduction techniques developed so far focus on one of the these two directions: reducing the number of generated mutants or reducing the execution cost [41]. Methods proposed for reducing the number of generated mutants focus on finding the subset of all possible generated mutants that would output a similar mutation score. Methods proposed for reducing the execution cost are focused on optimizing the mutant execution process.

Another disadvantage of mutation testing is the need for **manual analysis** for identifying equivalent mutants. It is possible that some mutations do not change the semantics of the code (e.g. swapping the order of two method calls or refactoring dead code) [41]. In this case, some of the mutants pass through all tests, just like the original program, because they behave exactly the same as the original program. This type of mutant is called an *equivalent mutant*. The equivalent mutants need to be identified because they have a direct

Criteria	Features	MuJava	PiTest	Bacterio	Judy	JavaMUT	JAVALANCHE	MAJOR	Jumble	Jester	Homaj
Technical	Maven support		yes			?			yes		
	Open Source	yes	yes	no		no					
	Active	2016	2016	2012	2014	?		2016			
Community support	Suggested for research	yes			yes	?		yes			
	Suggested for projects		yes			?					
	Git project	yes	yes			?					
Operator set	OO operators	Offut et al.	no	Offut et al.	Offut et al.	Chevalley et al.					Offut et al.
	Other operators	15	10	9	0	?					
	Option to select subset of mutants	no	yes	no	no	?	no	no			
Cost reduction Technique	Test selection		coverage	manual		?	coverage	coverage	manual		
	Parallelization		yes	yes	yes	?	yes				
	Equivalent mutants solution		no			?	yes				
	Fault injection method	source code & bytecode	bytecode	bytecode	bytecode	?	bytecode	bytecode	bytecode	source code	
	Detect strong mutation	yes	oracle	oracle	oracle	?	oracle	oracle	oracle	oracle	oracle
	Mutant schemata generation	yes	no			?	yes	yes			

Table 2.1: Java Mutation Testing Tools found my Zhu et al. [62] and own research

impact on the accuracy of the mutation testing output. The equivalent mutants also have to be ignored for further analysis since they do not convey any useful feedback. Unfortunately, this process is hard to automate as the general problem of determining that two mutants are semantically equivalent is undecidable [24].

2.6.3 Mutation Testing Tool: Pitest

Current mutation testing tools are **not compatible with continuous integration environment** and are also **inflexible** by nature, not allowing operator extensions or customizable constraints. This is based on a recent 2016 study that presents the practical usage of mutation testing in different studies. The authors, Zhu et al. [62] based their findings on 159 papers. Half of all the papers surveyed, used pre-existing open source mutation testing tools, while in 21 instances the authors implemented their own tools or manually created the mutants.

Inflexibility is not the only setback of current mutation testing tools. Most mutation tools do not implement cost reduction techniques. More than half of the papers surveyed by Zhu et al. [62], do not mention how they deal with one of the biggest problems mutation testing faces, labeling the seemingly similar faulty versions. The rest of the papers either ignored those cases or labeled them by manual analysis.

Since the mutation testing tools are inflexible, hard to setup and also do not implement cost reduction techniques, mutation testing is not very popular in practice. Nevertheless, in order to analyze continuous mutation testing in a experimental environment we selected the best pre-existing mutation testing tool. We looked at all mutation testing tools for Java we could find, most of them included in the survey of Zhu et al. [62]. We looked at compatibility, usage, maintenance and performance for each tool in Table 2.1. Based on this selection we decided Pitest [26] is the best fit for our continuous mutation testing experimental environment.

Pitest [26] is a very well maintained mutation testing framework for Java. It has good integration with build tools (e.g. Maven, Ant, Gradle), development environments (IntelliJ, Eclipse), and static code analysis tools (SonarCube). Pitest also presents all data computed in a structured and clear way. Nevertheless the output does not offer any filtering options

making the manual analysis very lengthy requiring a lot of system and mutation knowledge. However, the tool is being actively maintained and improved, creating a very helpful and responsive community.

Pitest is also fast [26], compared to other mutation tools because it applies the operators at bytecode level and implements test selection for each mutant based on coverage and execution speed. The current operator set of Pitest is considered feasible and practical on real world programs [26]. The fault seeding procedure is implemented using the ASM bytecode framework [2] and the BCEL framework [3] for local implementations. Test selection means that Pitest only runs the tests that cover the code that was mutated, so the collection of tests to be run is chosen based on coverage and on a timeout constraint to improve the speed of the process.

An important advantage of Pitest is that it makes a difference between tested and untested mutants. Most tools would consider both cases as survived mutants. However Pitest introduces a 3rd mutant label: NO COVERAGE. This information is crucial for the filtering steps in the continuous mutation testing process. Also Pitest checks for a green test suite. If any tests are being skipped or fail, Pitest will not analyze the project. This also means that if there is no coverage, Pitest will still analyze the system and report all mutants generated as not covered.

In principle Pitest‘s compatibility with Maven facilitates a CI compatible infrastructure. Nevertheless it does not provide an easy customization of the input on which to apply the mutation process. This makes Pitest unsuitable in its current version for large applications as shown by the empirical study conducted by Klischies et al. [44]. The limitations of the tool is also illustrated by data collected in this study.

Pitest has a new feature offered by the community which allows it to run on the last change. This means all files that contain at least one change will be analyzed. While still an early-stage feature with some problems, we will use it as our starting point for implementing continuous mutation testing.

Chapter 3

Proposed Solution

In a Continuous Integration(CI) environment, every time a change is made in the code base, the CI pipeline is being activated. This triggers the execution of the test suite, the results of which are automatically examined. If certain criteria are met by the results, the application is deployed. These criteria may be the following: all tests are passing,the code coverage is above a certain threshold etc.

Although mutation testing is used to some extent, it is not commonly used in CI environments. This is due to the major disadvantages of the mutation testing process - a) it is time consuming, b) it requires a lot of processing power as well as manual analysis, c) not all suggested fixes are highly relevant or there might be false positives. This is indicated by the design of mutation testing tools, since they are tailored towards analyzing complete systems.

The purpose of this study is to analyze whether mutation testing can improve the development process by integrating it in a CI environment. By running the mutation tests after each code change, we can continuously detect specific missing test cases. We will also detect faults within the tests of the code that was changed, which is most prone to introducing new bugs due to the modifications. By focusing only on the relevant code, not only do we reduce the resources required, but we also filter out less relevant suggested fixes for the system.

3.1 Process flow

We will refer to the process of applying mutation testing in a continuous integration (CI) environment as *change driven mutation testing* or *continuous mutation testing*. In this approach, a new step that generates mutants and runs tests on them should be added in the CI pipeline. In order to have a successful build, the tests have to kill all the mutants generated for the last committed piece of code. Generating the mutants and running the tests leads to a variety of scenarios. Each scenario will result in either a successful build or in the need to take an action that improves the process. All possible scenarios are shown in Figure 3.1.

The CI pipeline will always be triggered by a commit. Once a commit is made, our analysis focuses only on the change recently submitted. In order to be able to integrate

Label	Test Coverage	Mutant			Behavior Content Impact	Follow Up Action
		Killed	Survived	No Coverage		
1	NO	~	YES	x	~	Add Test Cases
2	~	NO	NO	NO	YES	Expand Operator Set
3	~	NO	NO	NO	NO	Improve Line Prefilter
4	YES	YES	NO	x	~	BUILD SUCCESS
5	YES	~	YES	x	~	Add Specific Missing Test Case

Table 3.1: Line Types in Changed Based Mutation Testing (”x” indicates it is logically impossible for that condition to take place at the same time as the rest of the conditions; ”∼” indicates that specific condition value makes no difference)

mutation testing in this analysis, the following requirements have to be met: a green test suite and changed code lines with the potential to affect the behavior of the system. In order to identify which code lines have this potential, different strategies may be implemented. However, the most basic rules are that the code has to contain an actual instruction (no comments or annotations) and it has to be part of the application logic implementation (no testing code). If this kind of lines have been changed, we are able to generate mutants. After we have identified the number of mutants for each line, the granularity of the analysis is at line level. We then apply the same analysis for each changed line.

Once the analysis granularity is at line level, we need to label each line with one of the 5 change driven mutation testing categories shown in Table 3.1. In order to infer the correct type, we look at the following criteria: 1) whether the line is covered by tests or not (test coverage), 2) the number of killed and surviving mutants for that line and 3) if the line has behavioral impact on the system. We also have to differentiate between surviving mutants for a line that is tested and one that is not.

Type-5: represents using mutation testing in the traditional approach, where the output is the mutation score for the test suite. This acts like a grade for the test suite performance and contains a set of surviving mutants that are either equivalent to the original system, or are indicators of specific missing test cases. The feedback potential of mutation testing is within the set of surviving mutants. However the surviving mutant set can only be created over tested code.

Type-4: represents the case of a perfectly tested system that would catch all faults. This is why when the code has test coverage and no surviving mutants, we consider the build a success and label that changed line with Type-4.

Type-3 and Type-2 when a line has no mutants created for it, it may mean one of two things. Either that line should not be analyzed (it is a comment or an annotation) or the current operator set is not compatible with that line of code. A perfect system would be able to pre-filter the lines that should be analyzed at the very beginning. However, the existence of Type-3 lines, gives a change of the system learning and improving its pre-filtering for the next commits.

Type-1: represents the case for the code that has no test coverage but could be mutated based on the syntax. Applying mutation testing on code that has no coverage results in a full set of surviving mutants. Even though this is accurate, the purpose of mutation testing is to check the quality of the test suite. Reporting survived mutants for code that has no tests would add noise to the manual analysis. Moreover, not all code that is not tested can be mutated. If the lines can be mutated, it means that they may contain faults. For this reason, we label a line as Type-1 if the line was changed, may be mutated, but has no test coverage. We consider this a critical area that should be tested.

3.2 Implementation

3.2.1 Infrastructure

Mutation testing was first proposed in 1971 [47] as a method to asses the effectiveness of a test suite. Since then, cost reduction techniques for mutation testing have been proposed, but they all focus on the method itself and not on how or when it is applied. In its current form, the available mutation testing infrastructure is not compatible with modern continuous software development. Mutation testing in its current usage is very time consuming, whereas CI pipelines require faster feedback, for instance the Facebook CI pipeline has a threshold of 10 minutes [52]. In order to study the effects of continuous mutation testing we created a prototype tool to conduct experiments, built on top of the available technology described in Chapter 2. Continuous mutation testing means the process should be triggered by a change, followed by the analysis of the actual changed code for the purpose of creating mutants. The way the code is analyzed has been described in detail in the previous section, following the flow in Diagram 3.1.

From the infrastructure point of view, the continuous mutation testing flow can be viewed as a pipeline that requires the following sub-systems:

1. versioning control system for the code base
2. difference analysis processor
3. mutation testing tool applied on a given code base
4. mutation testing output processor
5. a user communication channel for the improvement feedback extracted from the output

For the implementation of most of the required sub-systems, we reused available technologies: GitHub [9] for control versioning system, Pitest [26] for mutation testing and Operias [59] for difference analysis. We developed from scratch the systems which have not been already implemented. The change based mutation testing prototype tool which resulted is named OPi+. The name is based on the technologies used, Operias, Pitest, and

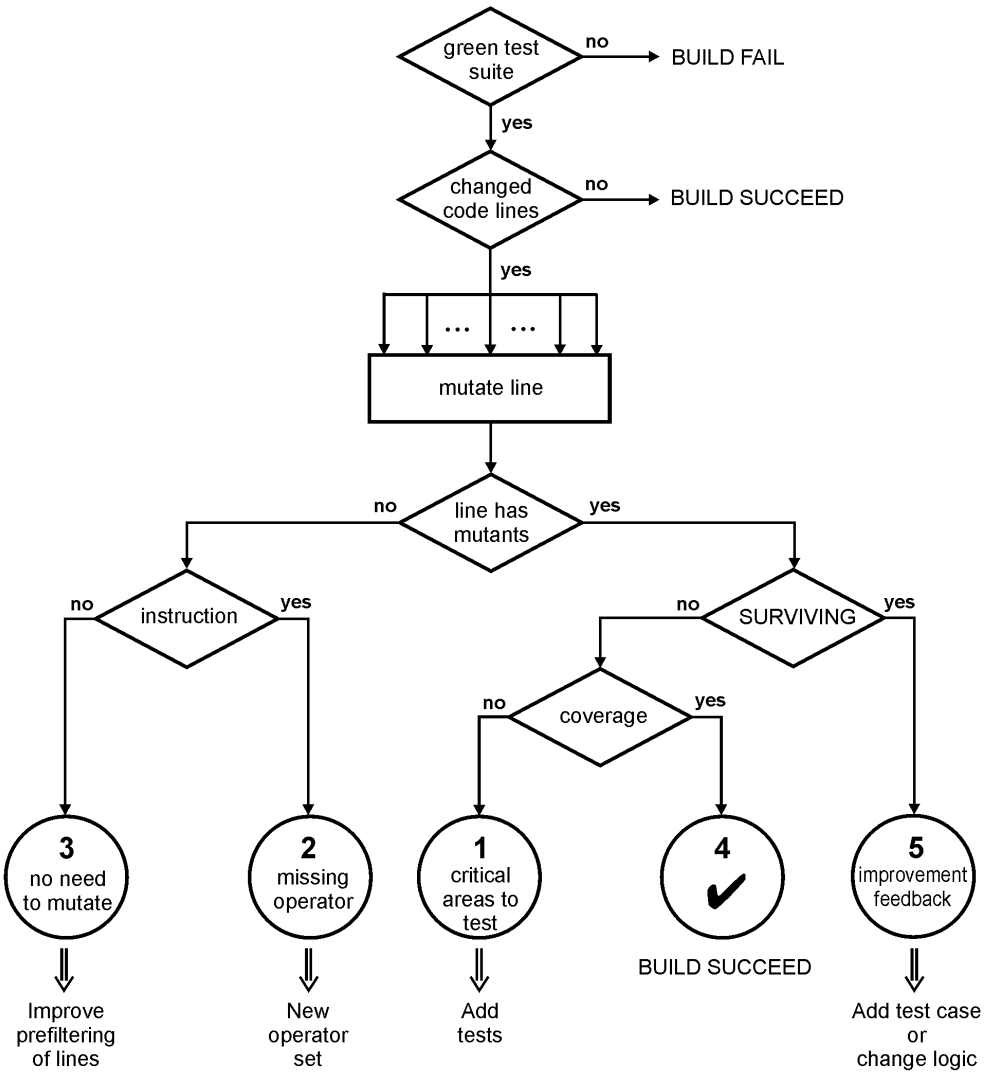


Figure 3.1: Continuous Mutation Testing Behavior Flow

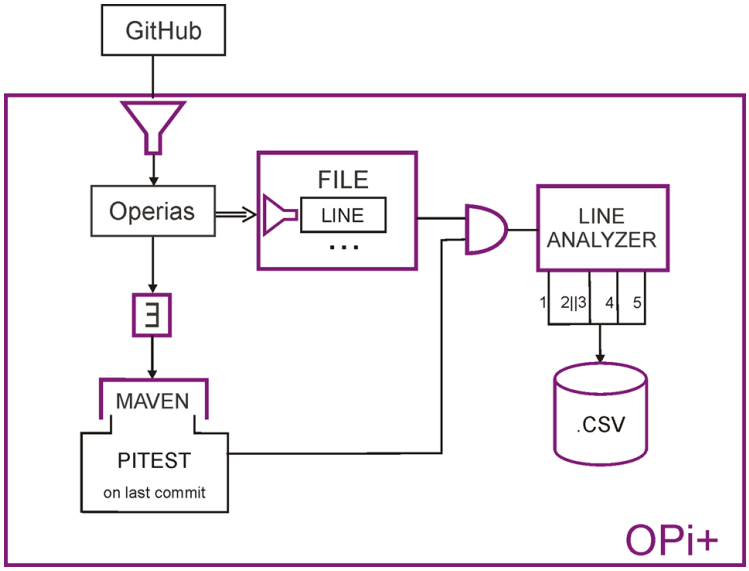


Figure 3.2: OPi+ Architecture

the extra computations we implemented. The project is hosted on Github[1]. The architecture of OPi+ is presented in the following sections and in Figure 3.2.

3.2.2 OPi+ Main Data Structures

Regardless of the technology used, continuous mutation testing requires that the code analysis must be made at line level. We consider that a simple mutation impacts only one line of code. Based on the theory that complex mutants are killed if the simple mutants that compose it are killed, each fault introduced by the mutation process requires changing only one line of code. Therefore, the core concept for change based mutation testing becomes a **Line**. The interaction between the main data structures is shown in the UML diagram 3.3. According to the continuous mutation testing flow, 3.1 we can detect 5 possible outcomes for any line, all described in the previous section. For each branch in the control flow graph, we instantiate a data structure in order to analyze it. It contains the following related information:

- the line number in the new version of the system
- the line change type (add or edit)
- the actual content of the line in the new version
- the total number of each type of generated mutants (surviving, killed, or not covered)
- a list of all mutants that survived (with name and description)

[1] https://github.com/ileontiuc/Continuous-Mutation-Testing

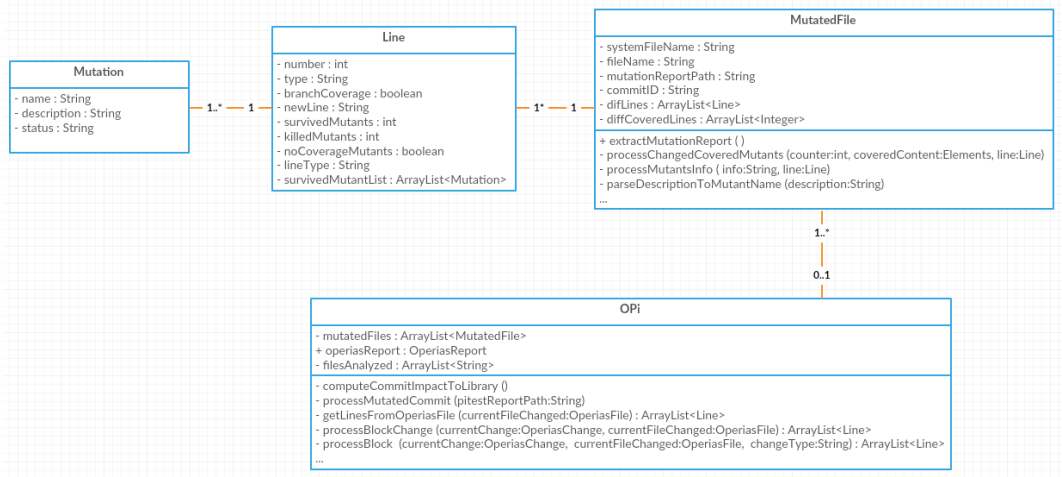


Figure 3.3: UML Representation of the Main Data Structures in OPi+

- whether this line has branch coverage
- the continuous mutation testing type that comes from processing this related data

A **Mutated File** is a changed file to which we also apply mutations at the difference points. A line is the core data structure for continuous mutation testing regardless of the technology used. However, the above information is influenced by technology. We chose Pitest as the mutation testing tool, which is capable of analyzing an entire file, not each individual line. Because of this, we instantiated a Mutated File for each mutation testing report generated for a file by Pitest. For each version of the file after applying the commit we record the following :

- full path of the file within the system
- name of file
- path to the file mutation report generated by Pitest
- the commit ID it belongs to
- a list of all changed lines within the file for the specific commit that are each represented by a Line

3.2.3 OPi+ Pre-filtering

Until now, applying mutation testing meant running the generation of mutants and testing them for the entire system. In order to integrate it in a continuous delivery process, we only need to apply it on the specific change or commit that triggered the CI pipeline. In our study, OPi+ looks at the entire commit history of a system in order to analyze the improvement this

integration can bring after each commit. However, not all commits are relevant for mutation testing. Therefore OPi+ narrows down the analysis area by incorporating two steps of pre-filtering, at commit level and at line level. Both pre-filtering steps are highlighted in the OPi+ architecture shown in Figure 3.2.

The **commit level pre-filtering** consists of selecting the commits that change only the code base for which we want to check the fitness of the tests. The commits contain at least one file for which the following conditions are true: 1) a `.java` extension, 2) it is placed in `src/main/java` and 3) it contains at least one added or updated line.

After selecting these commits, the second pre-filtering step changes the filtering granularity to line level. The **line level pre-filtering** is done after identifying all the lines that were changed (after executing Operias). At this stage, only the lines that can potentially change behavior are filtered. Mutation testing is applied only to code lines that can impact the systems behavior. Therefore, OPi+ filters out comments (starting with // /* or ending in */), logic unrelated lines (starting with @, import, package, {,},};,HTML tags) and empty lines.

In OPi+, we implemented this filtering process on top of JGit. We did not find another suitable tool for our requirements. The only commit filtering library we could find was Gitective [8], an outdated library built on top of JGit. Even after updating the dependencies to the latest JGit version, the filters provided by Gitective are very basic and inapplicable for continuous mutation testing (i.e. author information). The filters that do apply (i.e. .java extension) have poor performance time wise. In addition to the filters, the OPi+ implementation can also compute the number of change types for each file. Each file contains blocks of change. These blocks can either add, edit or delete code or have a path or file name changed. Each file is characterized in OPi+ based on the types of change blocks it contains.

3.2.4 Operias customization

Continuous mutation testing means the process is triggered by a change and the analysis is done on the change itself. In order to represent the change in OPi+, we use Operias [59], an open source tool that outputs the differences between two versions of a project, looking at the source code and also at the statement coverage. Operias outputs part of the information required by OPi+, such as the code that was changed and their branch coverage. Although Operias is designed to support the analysis of full pull requests, it can also be applied to compare arbitrary commits. It is this feature of Operias that we will be using. We consider the commit under analysis and the previous one, as the two different versions of the system which Operias needs to analyze.

In order to improve the OPi+ performance, we only trigger Operias if the version under analysis passes all test cases, even though Operias can run with failing tests. Also, by replacing the pull requests with commits, the tool generates its information more often in order to output reports, since the granularity is smaller. To alleviate the surplus of information, we cancelled the reports generation phase without altering the data processing algorithm. As soon as the analysis is done and differences are found, we cancel all further steps, take this data and transform it into OPi+ data structures such as Mutated File and Lines.

The core of Operias is structured into 3 different Threads. Two of the threads analyze the branch coverage for each version of the system by using Cobertura [4]. The third thread compares the source code of both versions, reporting the differences. For a successful Operias analysis, as well as a successful OPi+ commit analysis, all 3 threads require a successful run. If any of them crashes for any reason, we conclude that the commit may not be analyzed due to Operias limitations.

Setting up the correct environment to make Operias compatible with the OPi+ experimental process has proved to be challenging. Certain code change cases are not covered by Operias and treated with a `System.exit`. This happens in 19 places, out of which 17 affect the OPi+ process. We records all these cases as an Operias exception, which means that the commit can not be analyzed. One of these instances is used in the Configuration class, where Operias is parsing the command line arguments. If Operias can not parse an argument, it will stop the entire JVM. Nevertheless, OPi+ analyzes all commits, so one unparsable parameter set should not cancel the entire analysis process.

After extracting all the data from the threads, Operias merges the information from all of them, creating the core data structure called an Operias Report. The creation of this report can cause stack overflows that are treated the same as a thread failure. The commit that causes the stack overflow is ignored by Operias.

Once the Operias Report is successfully generated, OPi+ goes through all changes reported parsing them and creating the Lines data structures, clustered in `Mutated File` data structures. Operias generates multiple change instances for the same file. In order to filter out the duplicates, OPi+ parses the file names, recording only unique reported changes for each file. During this step we also compute the commit impact. A commit is a set of changes in the system spread across one or several files. Each file contains at least one block of code changed. A block represents a change, uninterrupted by code that is not being modified. Each block is formed out of at least one line. In the scope of OPi+, we are only interested in lines that are adding or editing source code. That is why we compute the impact of a commit as the total number of code lines added or updated.

$$\sum_{j=1}^{SelectedFiles} \sum_{i=1}^{Blocks} blocksize = CommitImpact$$

Once all changed files recorded by the Operias Report are transitioned to Mutated File, we trigger the mutation step. Pitest is only triggered if we have at least one file converted to a Mutated File.

3.2.5 Pitest Proxy

Continuous mutation testing is independent of specific mutation testing tools implementations. Each mutation tool, however, has different advantages and disadvantages. In order to make the OPi+ architecture modular, we decided to treat the mutation step as a black-box. The mutation black-box has as input the changes we want to analyze and as an output the mutant status for each of the lines analyzed.

The current OPi+ mutation black-box makes use of Pitest, one of the most popular and compatible mutation testing tools. Just as all the other mutation testing tools, Pitest was

designed to be run on the entire system. However the community contributed with a feature that allows to run Pitest on all files that were changed in the last commit [15].

In order to provide input to Pitest by OPi+, we designed a Pitest Proxy, which does the following actions. The new Pitest feature can only analyze the last commit made, so the proxy first clones the repository, changes the HEAD to the commit we want to analyze and then rebuilds the system. One of the advantages of Pitest is that the mutations are done at byte-code level, which is why the system needs to be built before providing the compiled version to Pitest. Pitest also requires certain dependencies to be met in the project settings which are ensured by the Proxy. We worked with Maven projects and the required dependencies are the valid SCM connection and a JUnit version higher than 4.5. SCM [13] stands for Software Configuration Management. It is a Maven plugin that lets users commit and update the system through Maven by connecting their Maven system to their own code control management system. For each commit we have to check the pom file of the changed system and update it accordingly. If the pom file is not compatible and we can not edit it, we label the commit as a commit that can not by analyzed by Pitest. Once the project is set, we can run Pitest on "the last commit", which due to the HEAD change is always the given commit.

Using only this Pitest feature and maven capabilities might seem enough for continuous mutation testing. In order for mutation testing to be usable in a continuous integration environment the process needs more steps. The current process does not minimize the manual analysis still required after the automatic part of mutation testing is done. This is due to the following problems:

1. **Fixed analysis scope:** Pitest analyzed a lot more code than the change itself. By using the new run on last commit feature of Pitest we still analyze the entire file than contains a change meaning a lot more code is being processed than required.

2. **No information filtering:** there is no filtering of the relevant information. All information processed by Pitest is being reported, such as code status, mutant status and tests run. This exhaustive information requires more time for the manual analysis.

3. **No prioritization features:** none of the information provided is being ranked. The analysis process is more time consuming if there is no prioritization of cases. The reports Pitest generates are well structured, but since they contain all the information regarding the mutation process, the useful information is hard to find.

The output of Pitest needs to be processed before reporting it through OPi+. If Pitest reports the build has failed, the system could not be analyzed and we consider that that commit cannot be analyzed due to Pitest and report it as such. If the build is a success, it does not necessarily mean Pitest generated a report for each file that was changed in the commit. The console maven output is logged in a temporary file. After parsing this file we infer whether Pitest did generate a report for the files we requested or not. Pitest may generate no report because of malfunctions that either do not detect any change or no mutants could be generated.

Once the Pitest report for a changed file of interest is generated, we parse it and extract the mutant information regarding the specific changed lines. This way, we properly simulate pure changed based mutation testing. From the Pitest report we extract the total number of each type of mutants and make a copy of all the surviving mutants per line. All this information is stored in the Mutated File data structure. This parsing step is based on the actual operator implementation of Pitest.

Setting up the correct environment for Pitest, running it and then processing the information by parsing the generated report is very inconvenient time wise, which why we use the process described above. We trigger the run of Pitest only in the cases we know we have lines that should be mutated. This is done by first pre-filtering the commits, then all the lines. We trigger Pitest on the current commit only if we detect that the commit contains at least one line of code with behavior change potential that was either added or updated.

The Pitest setup with the new feature had its own challenges. While making use of this feature we discovered several issues. Together with the Pitest creator, Henry Coles, we found a temporary solution and also reported an issue on the Pitest Issues page[2]. For certain projects, Pitest does not recognize any tests, so all generated mutants are labeled as not covered. Mutant labelling is a crucial functionality required for continuous mutation testing. Together with Henry Coles we found a temporary fix, by adding the test path in the command line.

3.2.6 OPi+ external dependencies

The OPi+ architecture ensures the communication between the subsystems described above. For this communication, third party libraries are used. These are represented as external facilities in the OPi+ dependency architecture, as shown in Figure 3.4 and are the following:

- JGit [11]: is used first in the commit pre-filtering stage. This library is built on top of the Git API. It can label each commit by its main change. For OPi+ we select commits labeled as ADD, MODIFY or RENAME and ignore DELETE and COPY. All the categories which contain code are worthy of further investigation. This library comes with behavior that is not always aligned with the needs of OPi+. For example, JGit labels some deletions as a file that is modified. Nevertheless, the OPi+ process correctly deals with this, and reports it as a system failure, yet proceeds with analyzing the rest of compatible code.

- Maven Apache Project [1]: is a Java library that can provide firing a Maven command from the JVM. We had to change the local running folder in order to execute Pitest with the feature that runs on a commit. This is an issue we discovered during the OPi+ development stage. Pitest's feature only runs if the pom file is in the local folder. We reported this issue on the Pitest Issue page [14].

- Jsoup [12]: is a Java library that provides easy parsing of HTML code. This library is used by OPi+ to analyze the Pitest report and extract relevant information.

[2] https://github.com/hcoles/pitest/issues/336

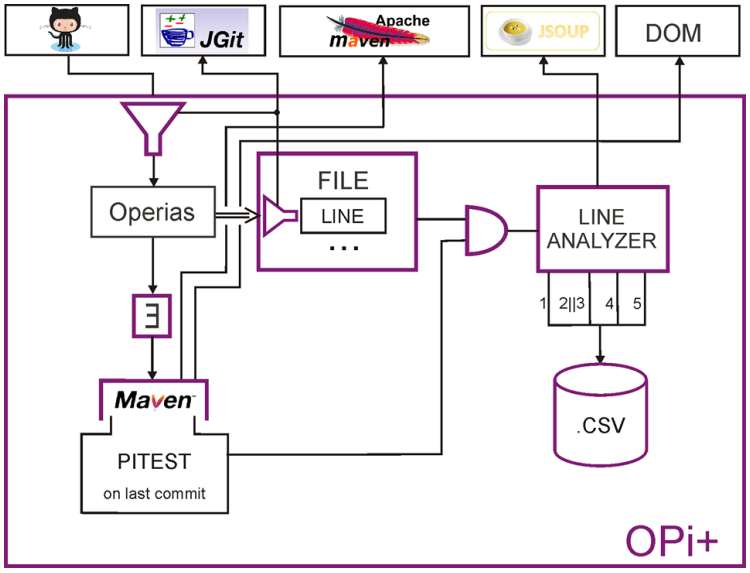


Figure 3.4: OPi+ Architecture 3.2 with External Dependencies

- DOM [6]: is a Java library used to analyze, change, add or delete XML elements. OPi+ makes use of this library in order to parse and update the dependencies in the pom file of the analyzed system.

3.2.7 OPi+ output computation

In order to infer the continuous mutation testing type for each changed line, we need information from both Operias and Pitest. Correctly pairing the information is done through path parsing and file name matching by Operias. This step is represented in the OPi+ architecture by the AND logic gate shown on Figure 3.2. If the data cannot be paired, it is ignored and reported as unparsable data due to Operias or Pitest.

The possible line types within the context of continuous mutation testing stay the same regardless of the specific technical implementation. The decision process in OPi+ has a slightly different decision tree than the continuous mutation testing behavior flow shown in Figure 3.1. The slightly altered decision tree is shown in Figure 3.5.

We first detect Type-2 or Type-3, that require either adding a new operator or better prefiltering. Both cases happen when we have no mutants generated on the line. The changes in the decision flow start from this point on since Type-2 or Type-3 is independent on any type of test coverage.

Pitest and Operias are based on different types of test coverage. Pitest is based on line coverage while Operias is based on branch coverage. More than this, Pitest uniquely reports 3 types of mutants: survived, killed and not_covered. Based on the Pitest implementation, the existence of no_coverage mutants implies there is no other type of mutant. The lack of coverage mutants means that there is no line coverage, which also implies no branch coverage. This case is labeled Type-1 and requires tests, since code with behavior change

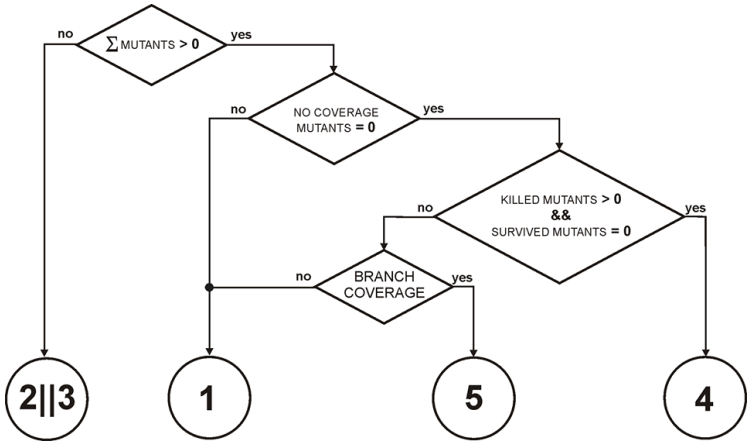


Figure 3.5: Continuous Mutation Testing Decision Flow in OPi+

potential is not checked.

In the case we have any mix of surviving and killed mutants, we can detect a special case. If all mutants are killed (meaning the number of killed mutants > 0 and the number of surviving mutants = 0), this code is very well tested. This specific scenario requires line coverage that is able to detect faults. In practice, we found branching statements that had line coverage but no branch coverage. Nevertheless, the partial branch coverage was able to detect the faults seeded by the mutation testing process.

If we have surviving mutants, then we also take branch coverage into consideration. Lack of branch coverage is the first indicator for missing tests cases. That is why we label lines with surviving mutants and no branch coverage as 1, which requires additional tests for the missing branches.

Chapter 4

Empirical Study

4.1 Research Questions

Mutation testing is a widely studied solution for assessing the fault detection ability of a test suite [41] and artificially generating faults [19]. Nevertheless, mutation testing operator sets do not scale well [57] and have a direct impact on the effectiveness of the method [53]. These shortcomings make current mutation testing practices incompatible with modern software development technologies such as continuous delivery.

The goal of this thesis is to explore how mutation testing can be applied to changes under analysis in a continuous integration setting. We particularly investigate the following:

1. How can we set up an infrastructure that allows for an efficient analysis of changes that are processed in a continuous integration server?
2. What are the costs and benefits of such an infrastructure when applied to realistic software changes?

The implications of this study can answer the following research questions:

- RQ1: How is the mutant generation, time and developer effort of mutation testing impacted in a continuous integration environment?
- RQ2: What are the implications of continuous mutation testing on the completeness of currently available operator sets?
- RQ3: What feedback information do the surviving mutants offer to developers in the context of a continuous integration setting?
- RQ4: Is there a correlation between surviving mutants and system characteristics such as: code coverage, churn and bad smells?
- RQ5: Can continuous mutation testing be used in the code review process?

To answer these questions, we will explore the histories, as visible through their commits, of a number of selected open source systems.

4.2 Project Selection

In this section, we present the projects we used in the empirical study to assess the costs and benefits of change driven mutation testing. We selected systems that have more than 400 commits that are valid for our analysis. Mutation testing can only be applied on code that is already tested, thus we selected projects with a code coverage value higher than 75%. The selected systems needed to also match the technical requirements of the tools we used. Both Operias and Pitest require as input a Maven based project with a passing test suite, so we needed only maven projects. Also, Pitest operator set applies to the Java language, thus we needed Java projects.

Selecting the projects that would meet all the above requirements was challenging. Our initial project filtering was popular Java Maven projects with a coverage higher than 56%. This lead to a subset of 22 projects. Out of this set, some systems could not be analyzed due to one of the following reasons:

- The system has a small number of classes
- The system is incompatible with Pitest: Spark[1], Retrofit[2], Evosuite[3]
- The system shows incompatibility with Cobertura through Operias: Auto[4], Apache Commons Collections[5], Guice[6]
- Operias goes into stack overflow when analyzing the system: ZXing[7]
- Operias enters an infinite loop when analyzing the system: Apache Commons Math[8]
- The system has unstable master versions(build fails on master branch and once we filter the failing commits out there are no more valid commits left to analyze): JUnit4[9], Apache Commons Lang[10], Joda-time[11]

For the empirical part of this study we will use three systems: JSoup, Apache Commons Compress and Apache Commons IO. These systems are compatible with the OPi+ infrastructure. We describe them in the following section.

JSoup[12] is a Java library for parsing HTML code. It is a popular open source system, actively maintained and hosted on GitHub. We selected this project because it is a

[1] https://github.com/perwendel/spark
[2] https://github.com/square/retrofit
[3] https://github.com/EvoSuite/evosuite
[4] https://github.com/google/auto
[5] https://github.com/apache/commons-collections
[6] https://github.com/google/guice
[7] https://github.com/zxing/zxing
[8] https://github.com/apache/commons-math
[9] https://github.com/junit-team/junit4
[10] https://github.com/apache/commons-lang
[11] https://github.com/JodaOrg/joda-time

Maven project with 62 classes. It also has a green test suite with 77% line coverage and 72% branch coverage. We analyzed the entire 988 commits, from 52 contributors, found in the history of the project.

Apache Commons Compress[13] is a Java API for working with several types of compressed files such as: ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200, bzip2, 7z, arj, lzma, snappy, DEFLATE, lz4, Brotli and Z files. It is a popular open source system, actively maintained and hosted on GitHub. We selected this project because it is a Maven project with 168 classes. It also has a green test suite with 83% line coverage and 72% branch coverage. We analyzed the entire 2206 commits found in the history of the project from 26 contributors.

Apache Commons IO[14] is a Java library for utilities used in develOPing IO functionality. It is a popular open source system, actively maintained and hosted on GitHub. We selected this project because it is a Maven project with 119 classes. It also has a green test suite with 90% line coverage and 86% branch coverage. We analyzed the entire 1941 commits found in the history of the project from 32 contributors.

4.3 Experimental Procedure

By using OPi+, we can simulate applying continuous mutation testing in the development process of a software system, by applying it to a series of earlier commits. We take all the commits made for that system and then we label each line, affected by each commit, with the labels described in chapter 3. After we have an overview of all the labeled lines, we analyze how the follow up actions, associated to the given labels, can help us find bugs.

The main purpose of OPi+ is to evaluate the cost and benefits of continuous mutation testing. We consider OPi+ an experimental environment because we repeat all the steps of a system‘s development process in chronological order: a commit is made, the system gets built and then tested. We then insert mutation testing as a last step, in each iteration. This way, we mimic a continuous mutation testing approach in a real development process.

4.3.1 Commit History Analysis

For each system we analyze, we consider all commits from the project's history. The OPi+ data structure that contains them is the **Commit File Library** described in Chapter 3. After applying the commit prefiltering from OPi+, we are left with commits that contain at least one code line changed in a .java file. For each commit, OPi+ retains the number of the included changed files. We can now categorize the commits in: large (more than 3 changed files), medium (2 changed files) and small (1 changed file) commits.

In order to maximize the efficiency of OPi+ we focus on analyzing only certain commits. We leave out a)branch merges and b)the commits whose content is incompatible with the OPi+ infrastructure (commits that lead to OPi+ crashes).

In GitHub, each merged branch is recorded as a commit. Because of their big impact on several files, merged commits end up being classified by OPi+ as a large commit, which is

not further analyzed. Change based mutation testing is an approach that targets one commit per iteration of the continuous integration development process. Before merging, the CI pipeline is triggered for each of the commits that make up all the changes included in the merge. Thus, analyzing a branch merge would be redundant and is outside the scope of this study.

If a commit cannot be analyzed, due to a crash of Pitest, Operias or OPi+, we label it as a crash. All crashes are recorded and described by the type, name of the exception and the cause. The crash types that can occur in OPi+ are as follows:

1. **A Pitest crash** is related to commits that cannot be processed by Pitest, such as:
   - could not find `pom` file
   - could not update dependencies in `pom` file
   - internal `Pitest` build failed
   - `Pitest` did not detect any change
2. **An Operias crash** means one of the three threads from Operias failed, which are described in Chapter 2. This means the change added by the commit could not be analysed.
3. **A System crash** contains any *Exception* that is not currently being handled by OPi+, such as:
   - missing path to mutation report
   - missing Pitest maven output
   - commit should have been pre-filtered (this means the filtering process should be improved. It is not an actual system crash)
4. **An Incompatible System Version crash** is when we cannot build that version of the system. These versions should not be present in the master branch. We also include very early versions of the system that do not pertain to the Maven conventions (i.e. no pom file).

4.3.2 Evaluation Data Gathering Process

In our continuous mutation testing analysis, each commit is broken down to line level. For each line, we record the type and, in the case of Type-5, the specific surviving mutants. We record all the lines in a project.csv file, along with the data described in Figure 4.1. The project.csv file will map every analyzed line of code to one row in the file, except for Type-5 lines. In this case, we have a separate row for each of the surviving mutants. For example, for a line with 3 surviving mutants, there will be 3 lines in the project.csv table. The surviving mutants information is computed by an OPi+ parser, based on the internal implementation of Pitest.

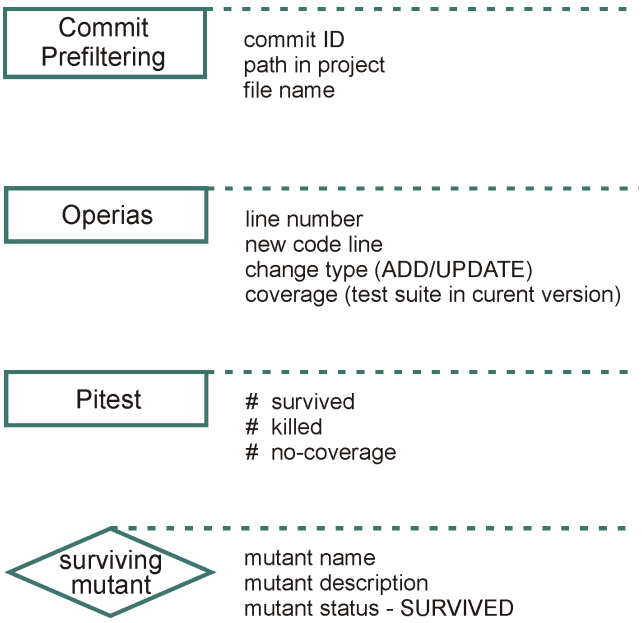


Figure 4.1: OPi+ Evaluation Data Gathering Process
Each text line represents a column in the data table.
Each box represents the step that generates the data for the specific columns

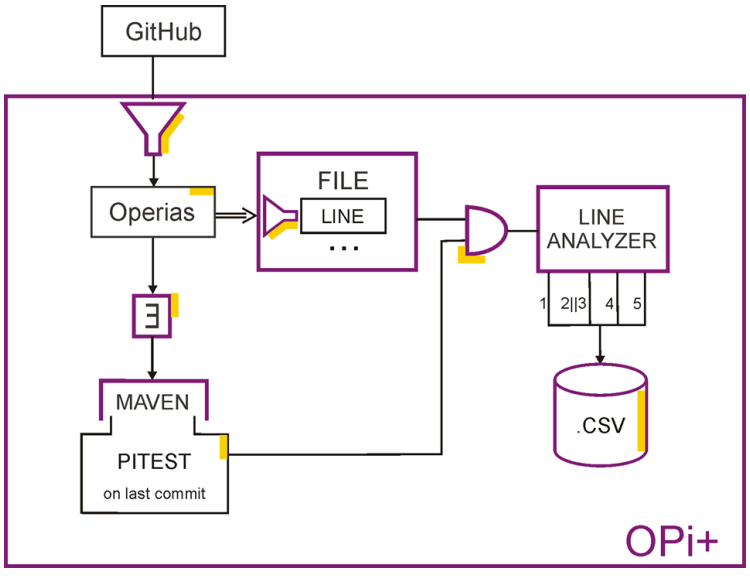


Figure 4.2: Data Extraction point in OPi+ Architecture

4.3.3 Additional Data Files

OPi+ logs its actions during certain stages of its execution. These are highlighted with orange over the architecture in Figure 4.2. All the data gathered is organized on areas of interests as follows:

- **Crash File:** contains all crashes with their detailed description (crash type, commit ID, cause)
- **Data File:** contains project name, link to repository, total number of commits, number of analyzed commits, total crashes per category, overall crash percentage, commits where pom file needed updated scm (described in Chapter 3) or JUnit dependencies, list of filtered commits (described by ID, mutation reports missing, file with mutation report), list of all commits (described by commit ID, how many files where submitted, how many lines should be analyzed, pre-filter status).
- **Log File:** contains a detailed step by step description of all the actions taken by OPi+
- **Commit Overview File:** contains a list of all the files for which Pitest did not generate a mutation report. These events are also recorded as a crash. We record commit ID, path to file in project and the file name.

4.4 In Depth Analysis of Output Types

When applying continuous mutation testing, each changed line of code can have one of the five labels described in Section 3.1. Each label has a different follow up action required, described also in Chapter 3. In order to study the impact these actions have in the development process, we take a different empirical approach for each label type.

4.4.1 Analyzing Type-3

A changed and analyzed line of code that is labeled with 2 or 3, means that there were no mutants created for it. Label 3 means the line should not be analyzed (like a comment line or annotation), and label 2 means that the available operator set is incompatible with this code. For Type-3 we suggest this information should be implemented in a better prefiltering step.

4.4.2 Analyzing Type-2

Once the prefiltering step is upgraded, we are left only with Type-2 lines. Since the current set of operators cannot be applied to these lines, we require more operators, since the changed lines have impact on the system‘s behavior. There have been studies conducted on operators [41]. However the way they were developed was to resemble known faults related to the language. In this case we decide on the needed operators the other way around. We see which code the current mutation operators leave untouched, and from that we propose new operators that we could apply. Based on this extracted information we can study the

implications of continuous mutation testing on the completeness of the used operator set, thus answering RQ2. Since OPi+ is using Pitest for the mutation testing step, making our study relevant since it is based on the most used operator set.

Due to the large number of Type-2 lines, we cannot manually analyze them so we apply data mining algorithms to cluster them. After a thorough analysis on the existing dataset, we find that the number of clusters should be bounded between 3 to 5. Hence, a silhouette analysis is attempted, in order to figure out the most suitable k-value. Unfortunately, the resulted cluster does not meet our expectations of instruction type based clusters. This leads us to investigate alternative approaches.

The data collected by OPi+ for a line is not numerical data, but contains the actual code line. Therefore we extract numerical text features based on TFxIDF. *Term frequencyinverse document frequency*(TFxIDF) [17] is a popular term weighting scheme in the field of information retrieval. This method tries to quantify the significance of a term in a document based on its frequency. In our analysis, we do not remove stop-words, which is a common approach to ignore terms with no meaning. Since the document we analyze is the code base, all terms could be relevant. After applying TFxIDF, we remove terms with highest frequency, terms that would not create a meaningful cluster, such as ”the”, ”a”, ”and”. These terms come from the comment lines, Type-3 lines we could not pre-filter. We generate a list of clusters that describe a specific type of code line that is not being analyzed due to lack of mutation operators.

We generate a list of all the top terms collected for all systems analyzed. We then group the terms based on their impact from a programming point of view. Since we analyze Java systems, we expect the terms to be related to object-oriented programming feature. Therefore we select the most relevant object-oriented mutation operators from literature, that match the terms we previously selected. More than this, we propose new operators or adjustments to existing ones, all part of answering RQ2.

4.4.3 Analyzing Type-5

A changed code line that has surviving mutants is the core of the mutation testing feedback. Analyzing these lines we can find specific missing test cases or even faults in the code logic. In principle, the existence of surviving mutants implies these lines are covered by tests but are not tested well enough. Nevertheless, a surviving mutant can also generate an equivalent system. Both the equivalence status or the actions that further need to be taken require manual analysis.

The continuous mutation testing approach would imply manual analysis on all lines labeled Type-5. Since in this study we evaluate the usefulness of this approach, we need to know: a) what type of information is stored in these lines, b) the distribution of each type and c) if we can detect a pattern for their spread. The research we conduct on this subset of lines is done at 3 different granularity levels as shown in Figure 4.3. The first one requires extensive and complex manual analysis, where as the rest are computed automatically. The levels of granularity at which we conduct the analysis are the following:

1. **Line level:** in order to detect relevant current problems unfixed in the system, thus

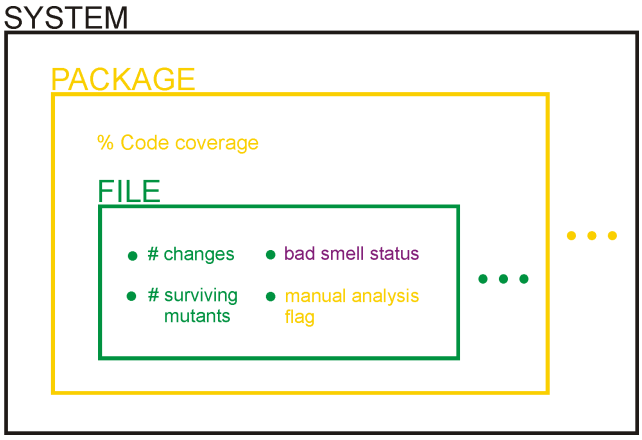


Figure 4.3: Label 5 in depth analysis

answer RQ3, we need to compare the version after the current analyzed commit with the current version of the system. There can be two possible scenarios after that:

- The analyzed line is no longer in the current version. This means it was previously deleted, leading us to investigate the reason for deleting it. It could be the code was irrelevant, however it could mean that a code line that has surviving mutants created problems for the system and it was changed later on. We also investigate how late the deletion was made.
- The analyzed line is still in the system. Due to the fact that changes were made between the commit analyzed and current version, the surviving mutant status might change. For this, we map the mutant status compared to the current version. A line with surviving mutants adds to the system technical debt by the liability it creates by not being tested well enough.

2. **File level:** is required for answering RQ4. We look at overall number of surviving mutants and the number of changes made to the file, over the entire commit history. We also apply the bad smell view computed for each file. A bad smell is an easy to spot, potential problem in the code base design [35]. Code that contains a bad smell is a prime candidate for refactoring. There are several bad smells presented by Martin Fowler in his book on Refactoring [35]. We detect bad smell classes by using InCode [49]. Based on the data distribution, we check for a correlation between changes, bad coding practices and the existence of surviving mutants.

3. **System level:** we can infer the mutant type to which the project is most sensitive. We generate a ranking of all surviving mutants found in the system commit history analysis. We also note the overall package test coverage. This information is part of answering RQ2.

4.4.4 Analyzing Type-1 and Type-4

For labels 1 and 4 we are only interested in the overall percentage. This information combined with the distribution of the other lines answers RQ1. Label 4 is for lines that are covered by tests, can be mutated and there are no surviving mutants. A very small percentage of this type of lines, found in a very well tested system, would mean the change based mutation testing approach is too fine grained and will require too many resources, similar to the traditional mutation testing approach. Since no further action is required for Type-4 lines, we look at the overall percentage of this case.

Label 1 means the changed line can be mutated, however it is not tested. Again we are only interested in the overall percentage and no further analysis is required because testing untested code is outside the scope of this study. Nevertheless, we explore whether continuous mutation testing has the potential of prioritizing the untested code, as this could render mutation testing even more valuable.

Chapter 5

Empirical Results

In this chapter, we present the results of the empirical study and thereby, we answer the research questions we raised in Chapter 4. The goal of this thesis is to *explore how mutation testing can be applied in a continuous integration setting in order to improve the code change analysis*. We particularly investigate the following:

1. How can we set up an infrastructure that allows for an efficient analysis of changes that are processed in a continuous integration server?

2. What are the costs and benefits of such an infrastructure when applied to realistic software changes?

The results are presented in the following sections.

5.1 RQ1: How is the mutant generation, time and developer effort of mutation testing impacted in a continuous integration environment?

5.1.1 Mutant Generation

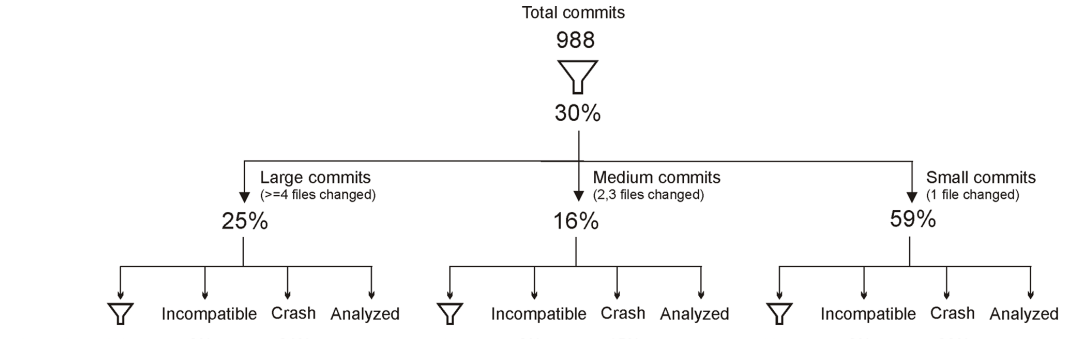
The processing power for mutation testing is required by the generation of mutants and the process of labelling them as killed or survived. Even after cost reduction techniques are applied, generating and running each mutant requires a considerable amount of processing power. In Table 5.1 we show the contrast between the generated mutants on the entire system and the generated mutants on the entire change history within the context of the project size, measured in classes. The Pitest columns represent the number of mutants generated on the last version of the system. The OPi+ columns represent all the mutants generated by processing all changes from the commit history. The results show an 80% reduction in the number of classes analyzed by OPi+ in comparison with Pitest. Also, continuous mutation testing in our study has a reduction of 91% in generated mutants. Out of these we have a 89% reduction in killed mutants and a reduction by 95% of surviving

mutants generated. The data collection process on which these results are based is presented in Sections 4.3.2 and 4.4.4.

Project	Size (classes)	Classes Analyzed		Generated Mutants		Killed Mutants		Survived Mutants	
		Pitest	OPi+	Pitest	OPi+	Pitest	OPi+	Pitest	OPi+
JSoup	62	49	23	15078	2197	9541	1955	5537	242
Commons Compress	168	168	21	28101	1233	17893	935	10208	298
Commons IO	119	109	26	10632	2051	8373	1706	2259	345

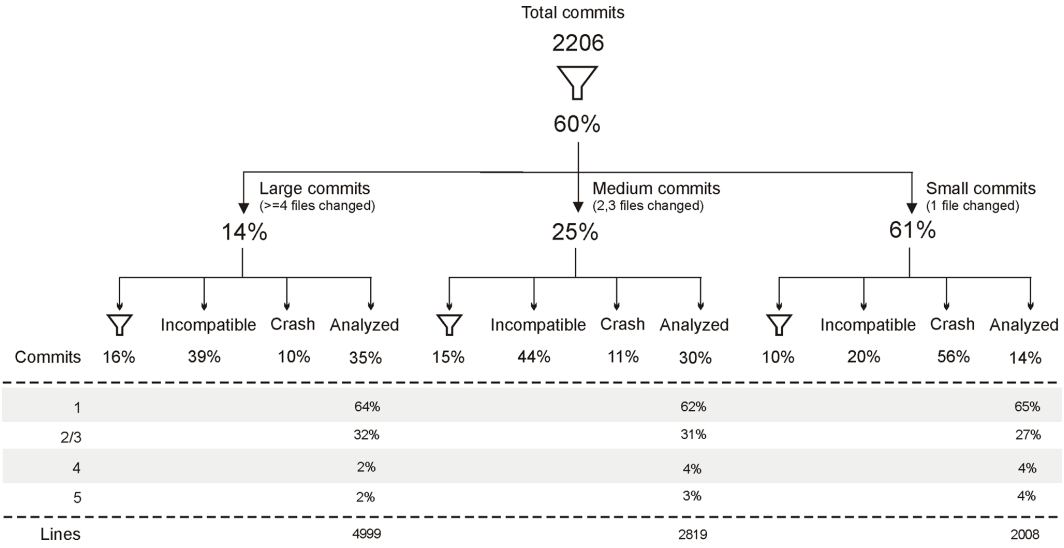
Table 5.1: Mutant Reduction Overview for Analyzed Systems

The way we selected which code to analyze is described in Chapter 3. Nevertheless, we give an overview of the selection process for each system in the Figures 5.2– 5.4. The diagrams follow the OPi+ filtering process. After selecting the commits that changed actual source code, we were left with 30%, 60% and 21% out of the total commits for the three different projects. We then split these commits in 3 size bins based on the amount of source code file changed. Then for each bin, we recorded the number of commits that should have been pre-filtered (false positives), are incompatible (the project cannot be successfully built from that commit version), caused a crash in the OPi+ environment, and the ones left to analyze. For all systems, on average, we were left with around 2000 lines of code to process. These lines are then labelled by OPi+ with one of the 5 continuous mutation testing line labels. On average, 38% of the lines required testing, 45% were not mutable due to the available operator set, 12% were thoroughly tested according to mutation testing standards and 5% required manual analysis. These statistics are based on the information provided in Diagrams 5.2– 5.4.



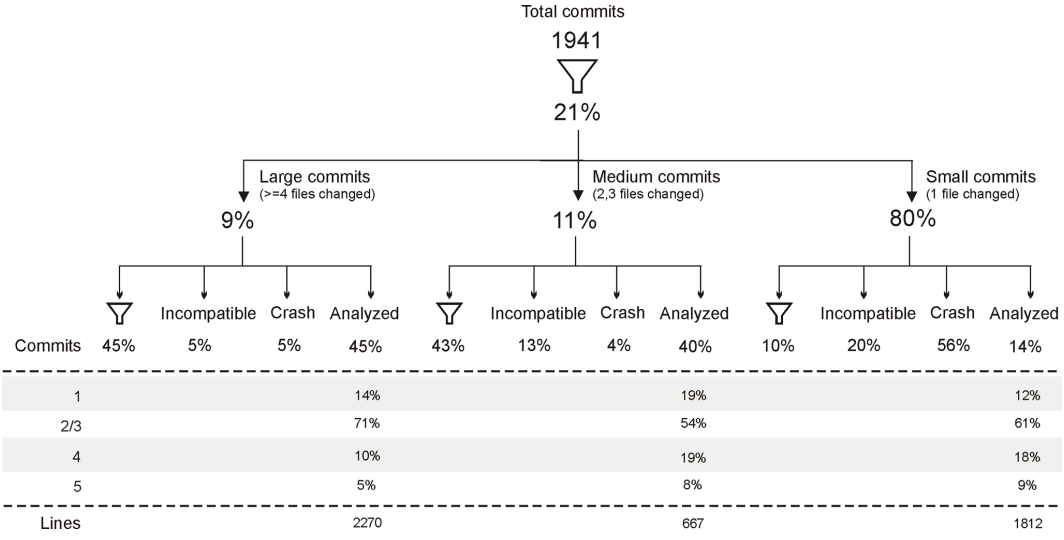
Commits	0%	0%	21%	79%	0%	0%	15%	85%	0%	0%	22%	77%
1			43%				42%				37%	
2/3			47%				46%				47%	
4			8%				10%				13%	
5			2%				2%				3%	
Lines			9993				1512				2582	

Table 5.2: OPi+ Commit Analysis for Jsoup



	Large commits (>=4 files changed)				Medium commits (2,3 files changed)				Small commits (1 file changed)			
Commits	16%	39%	10%	35%	15%	44%	11%	30%	10%	20%	56%	14%
1				64%				62%				65%
2/3				32%				31%				27%
4				2%				4%				4%
5				2%				3%				4%
Lines				4999				2819				2008

Table 5.3: OPi+ Commit Analysis for Commons Compress



	Large commits (>=4 files changed)				Medium commits (2,3 files changed)				Small commits (1 file changed)			
Commits	45%	5%	5%	45%	43%	13%	4%	40%	10%	20%	56%	14%
1				14%				19%				12%
2/3				71%				54%				61%
4				10%				19%				18%
5				5%				8%				9%
Lines				2270				667				1812

Table 5.4: OPi+ Commit Analysis for Commons IO

5.1.2 Time

The time required for mutation testing is comprised out of the time it takes to process the mutants (described in the previous section) and the time required for the manual analysis. Table 5.5 presents the overview of time required for the automatic processing part of mutation testing. The time it takes for Pitest to analyze the entire system is put in contrast with the time required to analyze the entire commit history (which includes running Pitest among other analysis). The history analysis is represented by the average time required for

a commit. This is linked to the time limit imposed by OPi+ on the Operias analysis. This time comparison is presented within the context of the project size computed in classes.

Even though Commons Compress and Commons IO have a similar size in Java core code classes, the time difference when running Pitest is due to a lot more timed_out mutants and warning such as ”unclosed ZipFile detected” or ”Unclosed BZip2CompressorOutputStream detected”. A time_out represents a test case that takes longer than expected when running a mutant. They are supposed to bypass infinite loops caused by mutants. These cases are reported separately but ignored by our study. However, the average time for running on a commit is much higher for Commons IO. The diff algorithm from Operias takes longer for Commons IO. To this end we had to change the Operias thread time limit from 1 minute to 10 minutes. All the cases that could not be analyzed by Operias brought this average time limit up.

Project	Size (classes)	Pitest on last system version (minutes)	Average time/commit (seconds)	Time limit imposed for Operias (minute/thread)
JSoup	62	60	60	1
Commons Compress	116	360	30	1
Commons IO	119	78	480	10

Table 5.5: Execution Time Overview of Systems Analyzed

5.1.3 Manual Analysis

All the commits analyzed were divided in 3 categories based on the size of the commit, with one file being the unit of measure. The commits labelled as Large were mostly branch merges. For JSoup, out of the 172 large commits, 37 are merge branches and 41 are pull requests. Even though a branch merge is logged as a commit within the GitHub repository history, the target of OPi+ is a traditional commit. We chose to continue the analysis only on small and medium commits since continuous mutation testing targets individual commits, not new branches. Because we discovered there is a correlation between churn and survived mutants analyzed by OPi+ (described in Section 5.4.1) we only manually analyzed the files that were changed 100 or more times, by looking for the number of mutants produced for each file. An overview of these files is given in Tables 5.6, 5.7 and 5.8. Here we have the number of mutants produced by Pitest (running on the entire, last version of the system) for each file in contrast with the number of mutants produced by OPi+ (running on all the commits). In the last 3 columns, we also show the number of lines in each file, the number of lines from that file analyzed by Pitest and by OPi+. The decrease of lines required for analysis is plotted in Figures 5.1, 5.2 and 5.3. In these figures we use a linear regression model type between the file size and number of lines analyzed by both Pitest and OPi+. We notice all p values for Pitest are <0.004, showing there is a correlation. This is expected since Pitest analyzes all possible code from a file. If the file size increases than the mutable code area also increases which in turn is more code that has to be analyzed by Pitest. However, OPi+ analyzes code based on the change not on the file size. If a file size

is large and is also changed frequently than we may see a correlation, which happens for Common IO. Nevertheless the other two systems have a p value >0.5 .

Project	File	Churn	Pitest Mutants			OPi+ mutants	Lines		
			Total	Killed	Survived		File	Pitest	OPi+
JSoup	TokenQueue	141	460	327	133	25	404	141	12
	Tag	367	203	82	121	67	371	119	24
	Elements	238	431	398	33	4	631	158	2
	Selector	241	72	59	13	4	175	41	4
	Document	167	289	156	133	11	562	160	6
	Element	316	872	714	158	35	1289	384	11
	Node	147	450	361	89	8	703	233	6
	DataUtil	100	304	235	69	6	255	105	5
	HttpConnection	546	1151	244	907	2	1114	541	1
	W3CDom	103	128	114	14	4	173	71	2
Average		236.6	436	269	167	16.6	567.7	195.3	7.3

Table 5.6: JSoup: Manual Analysis File Overview

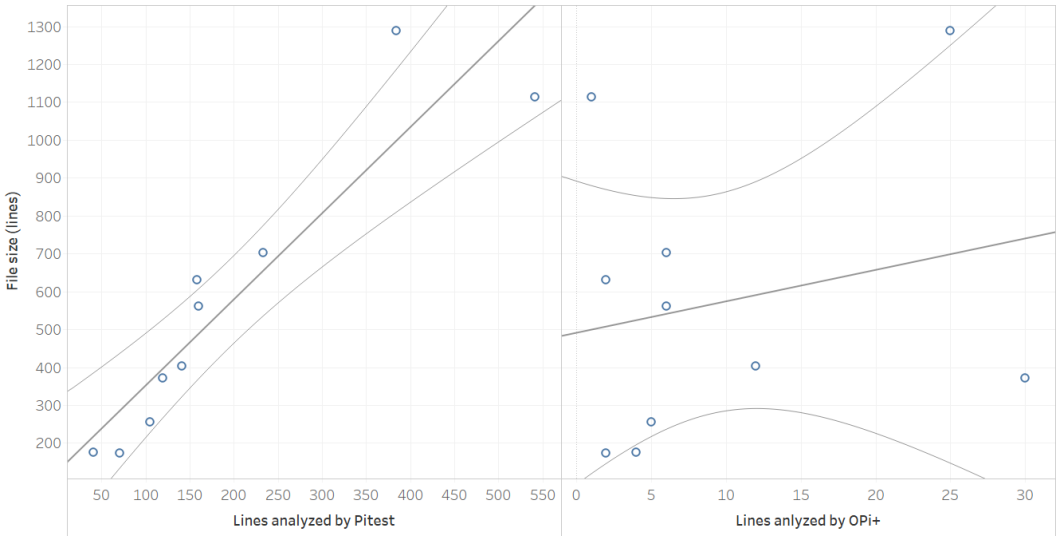


Figure 5.1: Jsoup File Size vs Pitest and OPi+ analyzed lines (p=0.0001592; p=0.540378)

Project	File	Churn	Pitest Mutants			OPi+ mutants	Lines		
			Total	Killed	Survived		File	Pitest	OPi+
Commons Compress	CpioArchiveOutputStream	117	544	345	199	0	563	218	0
	TarArchiveEntry	152	798	314	484	32	1259	304	16
	TarArchiveInputStream	306	545	129	416	9	766	238	8
	TarArchiveOutputStream	234	555	236	319	15	703	238	14
	TarUtils	156	515	126	389	55	612	169	27
	X7875_NewUnix	119	148	18	130	8	353	71	7
	ZipArchiveInputStream	308	1113	480	633	6	1136	417	4
	ZipArchiveOutputStream	323	1403	647	756	0	1258	532	0
	ZipFile	359	752	283	469	38	1258	379	16
	BlockSort	406	1247	650	597	11	1082	424	6
	BZip2CompressorOutputStream	1032	1339	474	865	0	1334	621	0
Average		319.27	814.45	336.55	477.91	15.82	938.55	328.27	8.91

Table 5.7: Common Compress: Manual Analysis File Overview

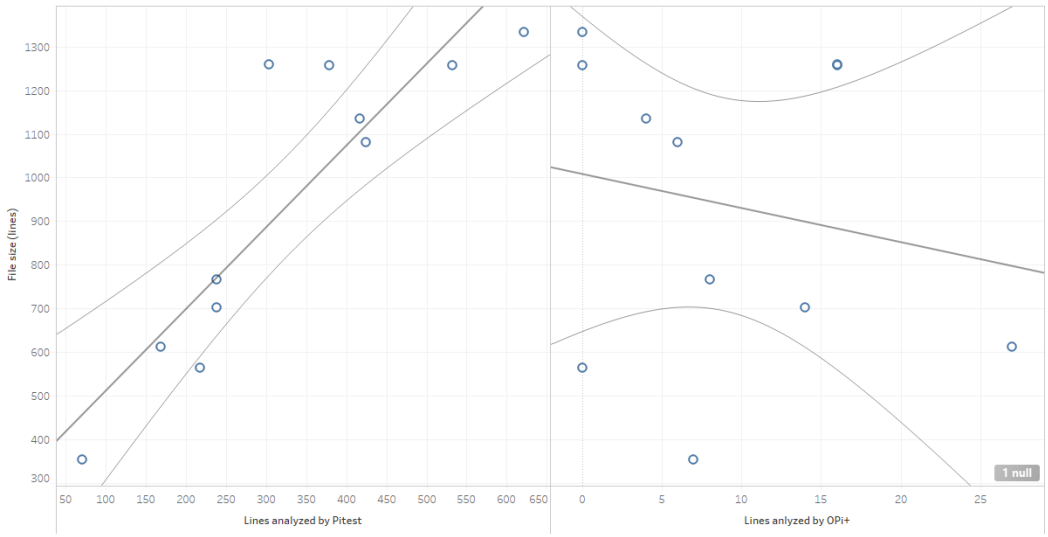


Figure 5.2: Common Compress File Size vs Pitest and OPi+ analyzed lines (p=0.0002855; p=0.569648)

Project	File	Churn	Pitest Mutants			OPi+ mutants	Lines		
			Total	Killed	Survived		File	Pitest	OPi+
Commons IO	IOUtils	442	981	218	763	49	3246	457	29
	FileUtils	376	1726	450	1276	65	3109	668	40
	ReversedLinesFileReader	219	118	14	104	22	366	52	9
	Tailer	131	179	75	104	14	557	128	13
	WindowsLineEndingInputStream	108	96	13	83	18	134	43	12
	XmlStreamReader	242	555	139	416	64	796	204	23
Average		253.00	609.17	151.50	457.67	38.67	1368.00	258.67	21.00

Table 5.8: Common IO: Manual Analysis File Overview

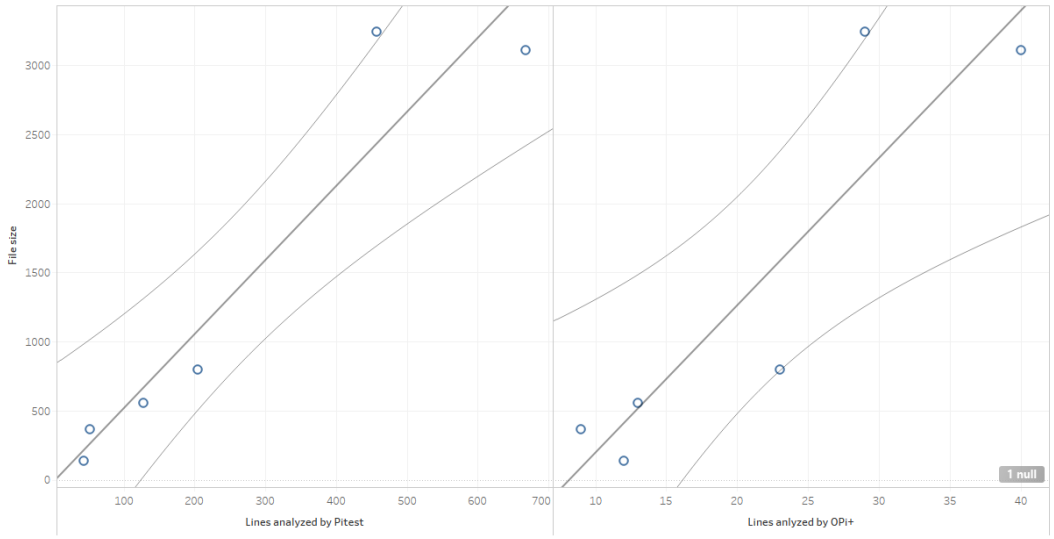


Figure 5.3: Commons IO File Size vs Pitest and OPi+ analyzed lines (p=0.0036486; p=0.014331)

Each file that we manually analyzed is described in Appendix A. An overview of this analysis is given in Table 5.9. Here we contrast the average number of mutants generated by OPi+ with the average number of mutants generated by Pitest over all the manually analyzed files. We present this within the context of the average churn of the files analyzed and the overall churn and commits of the systems.

Project	Churn	Commits	Average values from manually analyzed files							
			Churn	Pitest Mutants			Mutants	Lines		
				Generated	Killed	Survivied	OPi+	File	Pitest	OPi+
JSoup	4188	988	236	436	269	167	17	568	195	7
Commons Compress	4956	2206	303	777	311	465	15	878	328	9
Commons IO	2612	1941	253	609	151	458	39	1368	259	23

Table 5.9: Developer Manual Analysis Overview for Analyzed Systems

Mutation testing requires all surviving mutants to be manually analyzed. The number of surviving mutants OPi+ proposed for manual analysis is increasingly smaller than the surviving mutants generated by Pitest for the last version of the same file. We plot the number of surviving mutants generated by Pitest and the number of surviving mutants that were proposed by OPi+ to be analyzed in the entire commit history of the systems in Figures 5.4, 5.5 and 5.6. There is no correlation between these numbers since larger files do not necessarily imply more changes. Nevertheless we can see most data point are in the left side of the plots, which shows that OPi+ scales very well for large files. However, for Commons IO, we have a p value of 0.0001 which implies there a significant correlation between number of survived mutants generated by OPi+ and Pitest. This is due to the fact that in the case of this system large files tend to also be changed a lot. Nevertheless most

files are still on the left side of the plot, and even for large files we still have a high reduction in number of generated mutants.

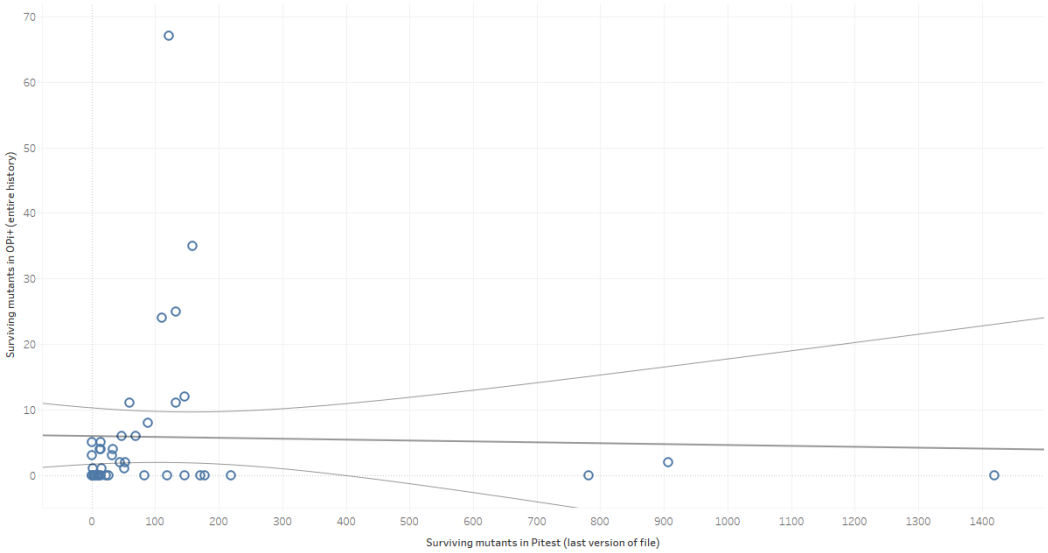


Figure 5.4: JSoup: Surviving mutants generated by OPi+ from the entire change history vs survived mutants generated by Pitest for the last version of the same file (p=0.848864)

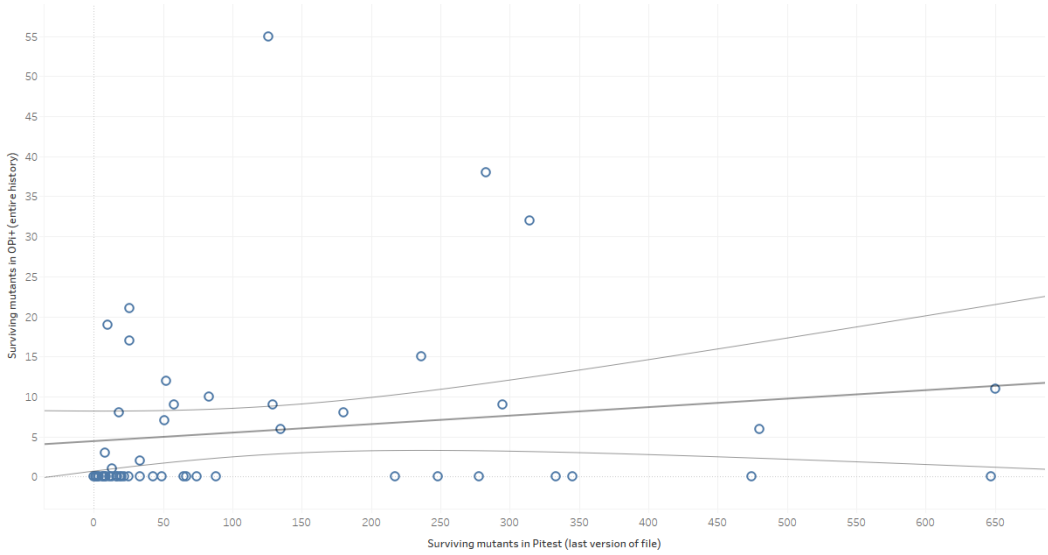


Figure 5.5: Commons Compress: Surviving mutants generated by OPi+ from the entire change history vs survived mutants generated by Pitest for the last version of the same file (p=0.252013)

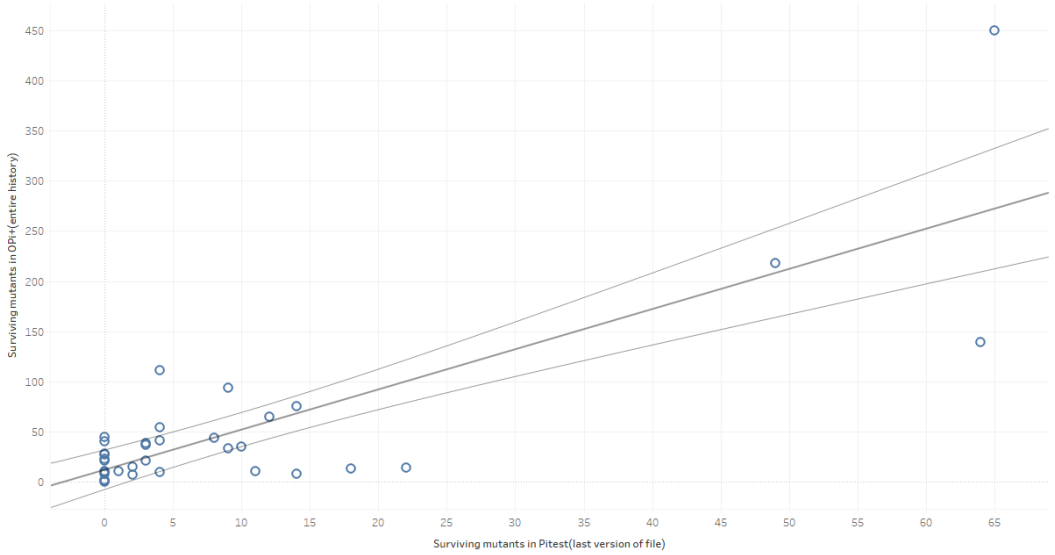


Figure 5.6: Commons IO: Surviving mutants generated by OPi+ from the entire change history vs survived mutants generated by Pitest for the last version of the same file ($p < 0.0001$)

5.2 RQ2: What are the implications of continuous mutation testing on the completeness of currently available operator sets?

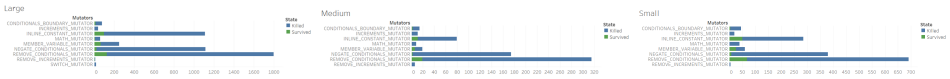
The overview of the OPi+ analysis from Figures 5.2– 5.4 shows that on average 45% of code lines changed were not mutable by the used operator set, as shown in the Type 2/3 line data. This means we are not able to reproduce faults and seed them into the system due to the incompatibility of the operators set. This data was collected and analyzed as described in Sections 4.4.2 and 4.4.3.

The main terms that indicate types of instructions that can not be mutated are presented in Table 5.10. The data is presented for each system in decreasing order of the frequency. Each term indicates to a type of language specific instruction that is popular in the analyzed code base. Nevertheless we can group all these popular terms into groups that represent the type of impact or role they have on the system, such as: modifiers, system specific classes or system specific method calls. Once these popular term based groups are detected we can more easily pair each group to a mutation operator. This way we identify important unimplemented mutators.

JSoup		Commons Compress		Commons IO	
Token	TFIDF	Token	TFIDF	Token	TFIDF
string	208.17	java	730.89	java	689.03
public	189.50	compress	243.89	src	230.04
return	160.61	main	243.72	io	229.58
private	119.45	commons	243.67	apache	229.43
element	95.93	apache	243.52	commons	229.43
elements	91.69	org	243.52	main	229.43
static	86.59	src	243.52	org	229.43
createinline	79.20	false	240.42	add	213.77
else	78.37	archivers	235.34	true	203.43
new	75.81	add	226.29	ioutils	198.71
createblock	71.48	update	184.40	input	195.02
final	70.91	zip	176.02	reversedlinesfilereader	190.36
tag	67.91	tar	138.73	false	185.48
node	62.68	true	123.60	update	168.97
this	61.46	compressors	120.98	fileutils	156.59
boolean	50.10	bzip2compressoroutputstream	116.53	public	106.76
int	49.68	final	114.53	final	99.45
code	43.51	zipfile	113.71	xmlstreamreader	84.60
the	42.50	int	110.20	file	79.19
td	27.84	bzip2	99.36	static	78.14

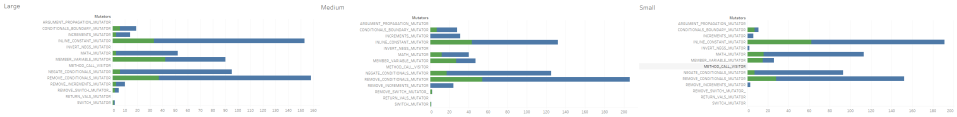
Table 5.10: Most Important Words computed via TFxIDF method from all Type-2 and Type-3 OPi+ lines

For each system we also plot the usage of each operator on the entire change history. The operators are from the Pitest mutator set and we show the number of killed and survived mutants of each type for each size group of commits in Figures 5.7, 5.8 and 5.9.



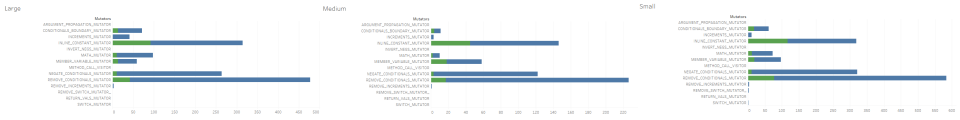
(a) Large Commits (b) Medium Commits (c) Small Commits

Figure 5.7: Surviving Mutants in JSoup Commit History



(a) Large Commits (b) Medium Commits (c) Small Commits

Figure 5.8: Surviving Mutants in Commons Compress Commit History



(a) Large Commits (b) Medium Commits (c) Small Commits

Figure 5.9: Surviving Mutants in Commons IO Commit History

5.3 RQ3: What feedback information do the surviving mutants offer to developers in the context of a continuous integration setting?

The manual analysis described in Section 5.1.1 also contained a comparison between the number of survived mutants identified by OPi+ and the number of survived mutants identified by Pitest from the last version of the system. We did this comparison because the last version of the system is closest to what the system is supposed to be. With this perspective we can infer how the existence of surviving mutants impacts the system. The entire analysis procedure is described in Section 4.4.3. We discovered 6 possible outcomes for the evolution of a line containing a surviving mutant within the system‘s history. The distribution of these cases is plotted in Figures 5.10, 5.11 and 5.12.

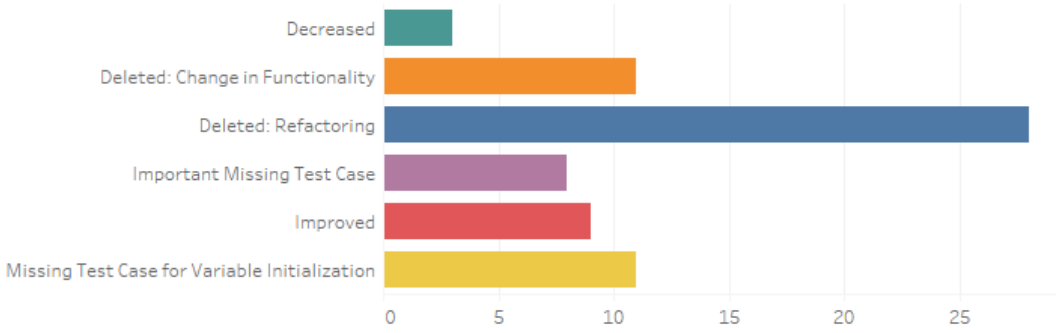


Figure 5.10: Distribution of Type-5 lines in JSoup

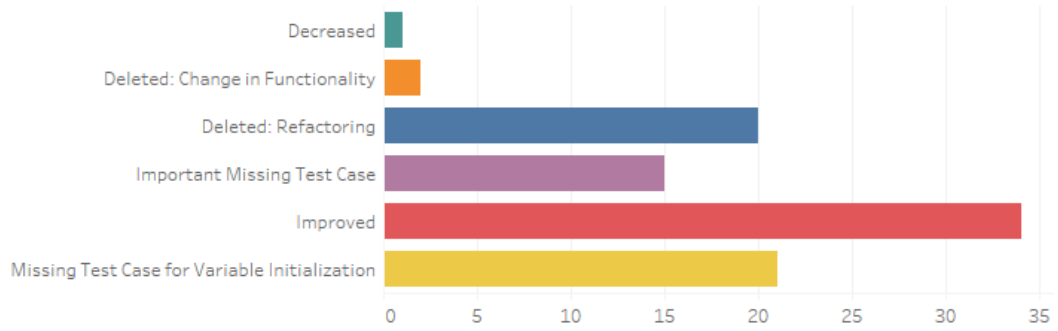


Figure 5.11: Distribution of Type-5 lines in Commons Compress

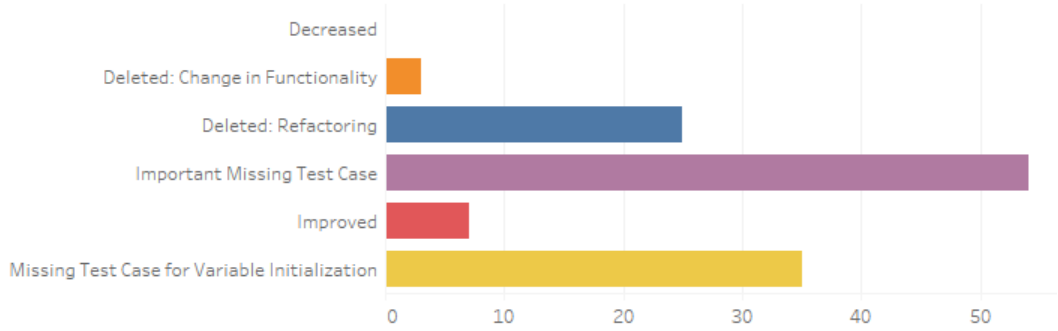


Figure 5.12: Distribution of Type-5 lines in Commons IO

Out of this analysis, we observed 2 possible states: either the line was deleted or is still in the system. If the line was deleted, there can be two possible causes: refactoring or change in functionality. If the line still exists, we also compare the line's mutants status. In this case, we define the following cases:

- **Missing Test Case for Variables:** the missing test case refers to a local variable or field initialization not being tested
- **Important Missing Test Case:** this is an important missing test case that could collect technical debt (not included in previous case)
- **Improved (all mutants are killed):** due to new test cases being added in the mean time, the line mutation testing status has improved
- **Decreased (all mutants become not covered):** due to an unknown reason tests that previously covered this area are deleted

5.4 RQ4: Is there a correlation between surviving mutants and system characteristics such as: code coverage, churn and bad smells?

To the best of our knowledge, no other study researched a correlation between surviving mutants and specific system characteristics. Even though a system can be characterized in many different ways, based on the limitation of this study we chose to look at churn, code coverage and bad smells. The specific steps of the empirical approach are described in Section 4.4.3. In what follows, we reflect on the identified correlation between surviving mutants and the aforementioned characteristics.

5.4.1 Code Coverage

Mutation testing and code coverage are both methods that asses the quality of a test suite. Therefore we would expect a decreasing number of surviving mutants for increasing code coverage. Nevertheless, the methods are very different by nature. Mutation testing checks the fault capability of a test suite. Code coverage checks whether the code is being executed by a test. Since there are studies that showed high test coverage does not necessarily imply a good test suite [39] [40], we expect to have no correlation between code coverage and number of surviving mutants.

We computed the branch coverage for the current version of Jsoup using EclEmma [7], based on JaCoCo [10]. We plot the relationship between branch coverage, churn and number of surviving mutants created by Pitest and OPi+ in Figures 5.13, 5.14 and 5.15. We make bins of branch coverage intervals on x-axis for all the files on the y-axis. The colour of each bin is based on the number of survived mutants generated by Pitest in the first plot and the survived mutants generated by OPi+ in the entire history analysis. We can notice that high coverage is not an indicator of good tests, since for classes that have full coverage we have a number of surviving mutants from all ranges. The different range in the number of surviving mutants is also kept when looking at mutants generated by Pitest.

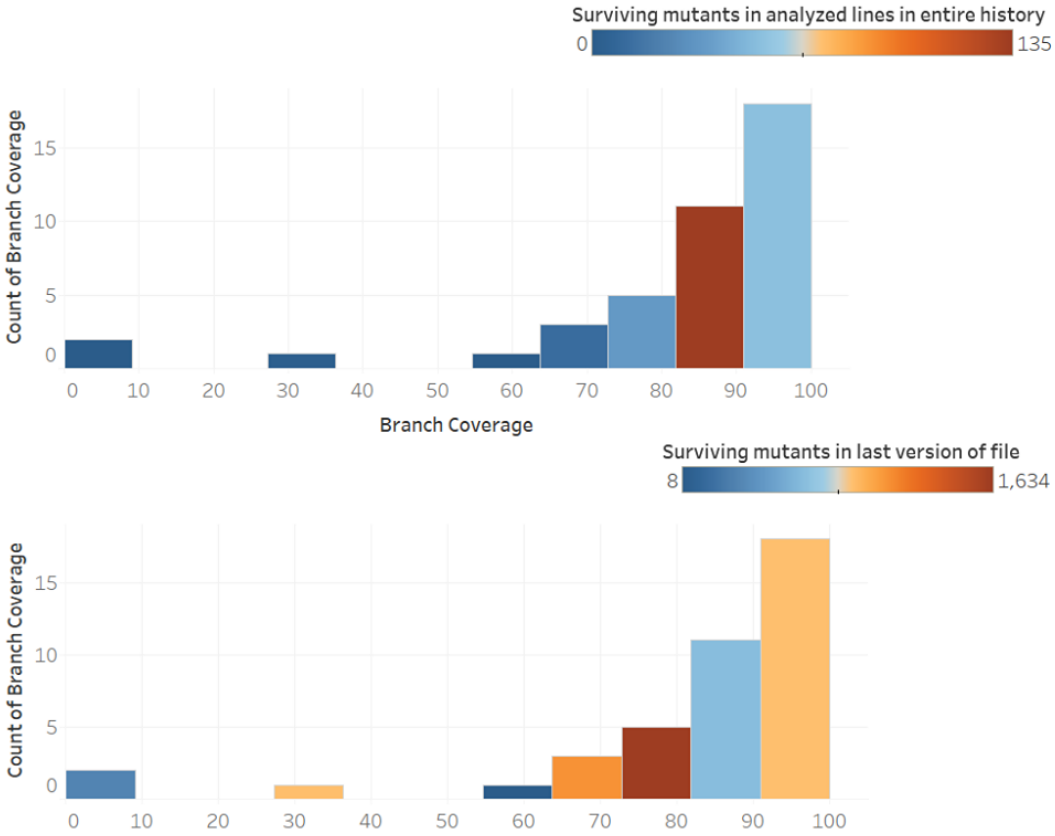


Figure 5.13: Jsoup Branch Coverage for each Class vs Surviving Mutants found by OPi+

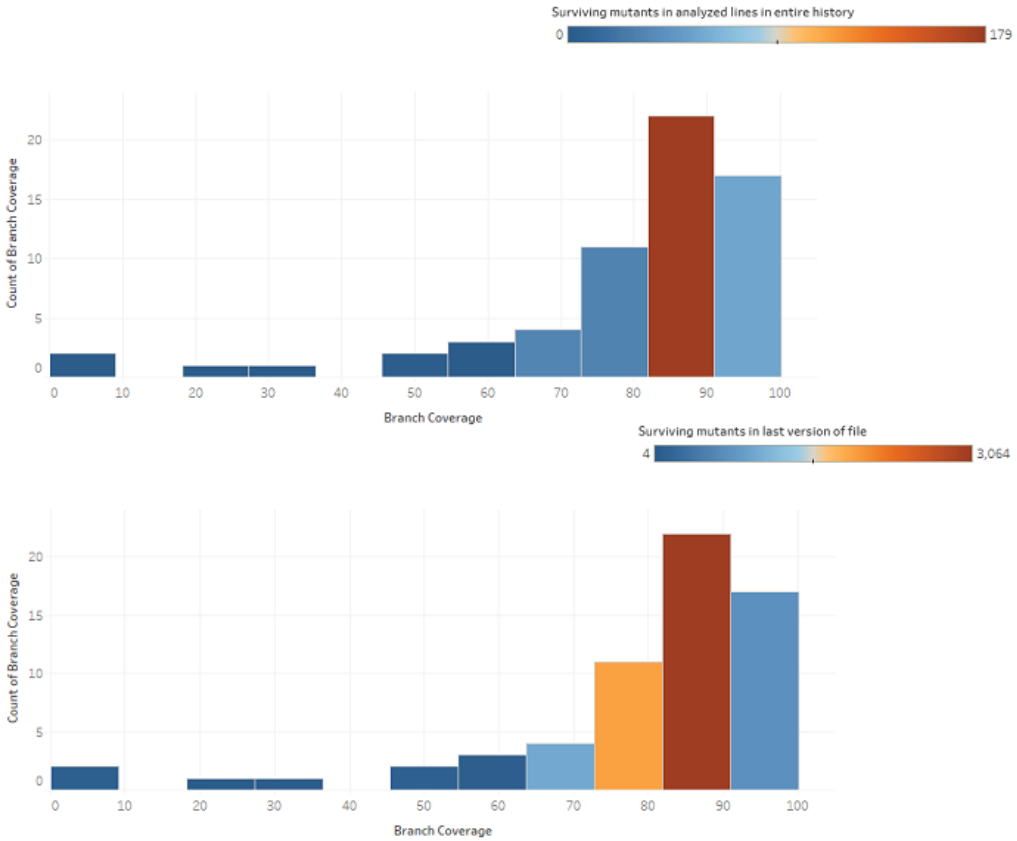


Figure 5.14: Commons Compress Branch Coverage for each Class vs Surviving Mutants found by OPi+

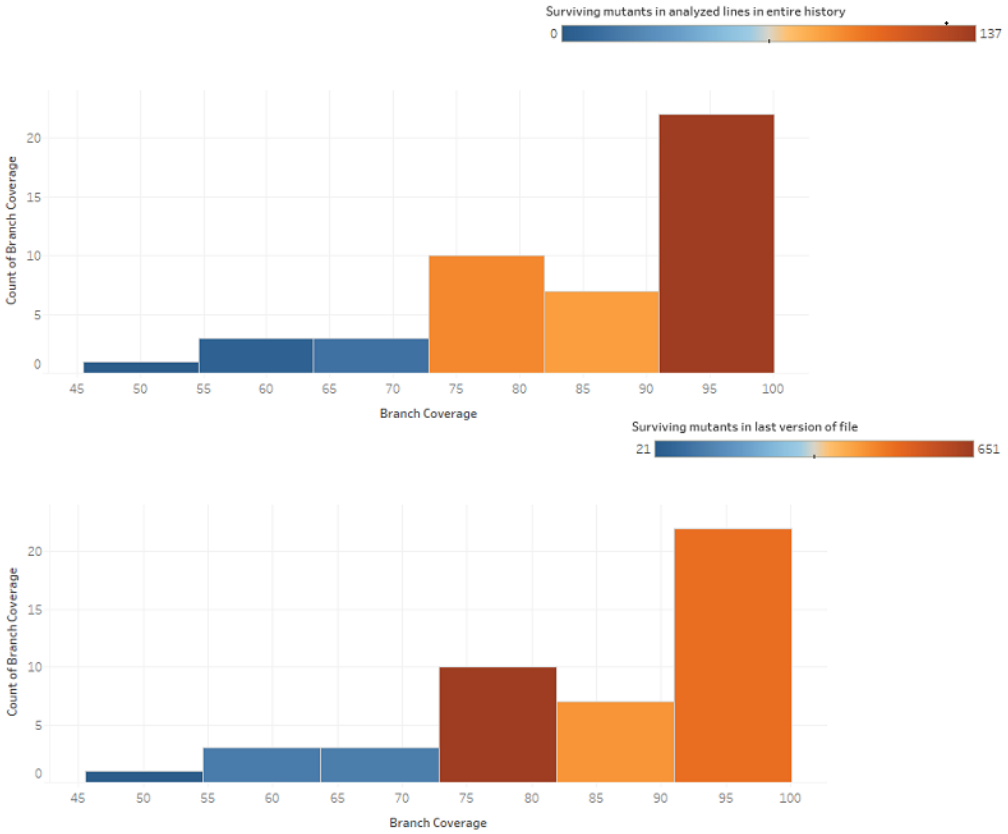


Figure 5.15: Commons IO Branch Coverage for each Class vs Surviving Mutants found by OPi+

5.4.2 Churn

Continuous mutation testing is a change driven approach. This means surviving mutants are only generated for changed code. Therefore, we expect a correlation between the number of survived mutants and the number of times a file has been changed. We plot the correlation between changes made for each class in the entire history of the project and the survived mutants found by OPi+ in that class. This is shown in Figures 5.16, 5.17 and 5.18.

We obtain the expected correlation between churn and number of survived mutants required by continuous mutation testing for 2 out of the 3 systems analyzed, with a p value $<$ 0.002. However for Commons Compress we do not have a significant outcome. Nevertheless for this particular system, we had the highest percentage of non analyzable commits, meaning we do not have enough data to infer this correlation.

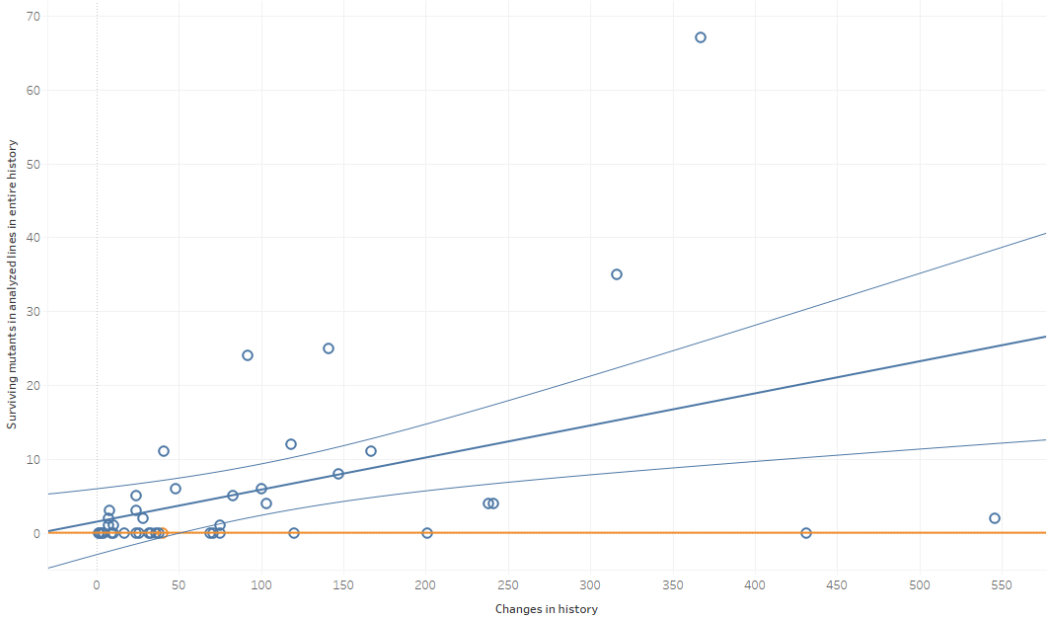


Figure 5.16: JSoup Churn vs Survived Mutants computed with OPi+ for each class (p=0.0029194), highlighted with orange Bad Smell Classes

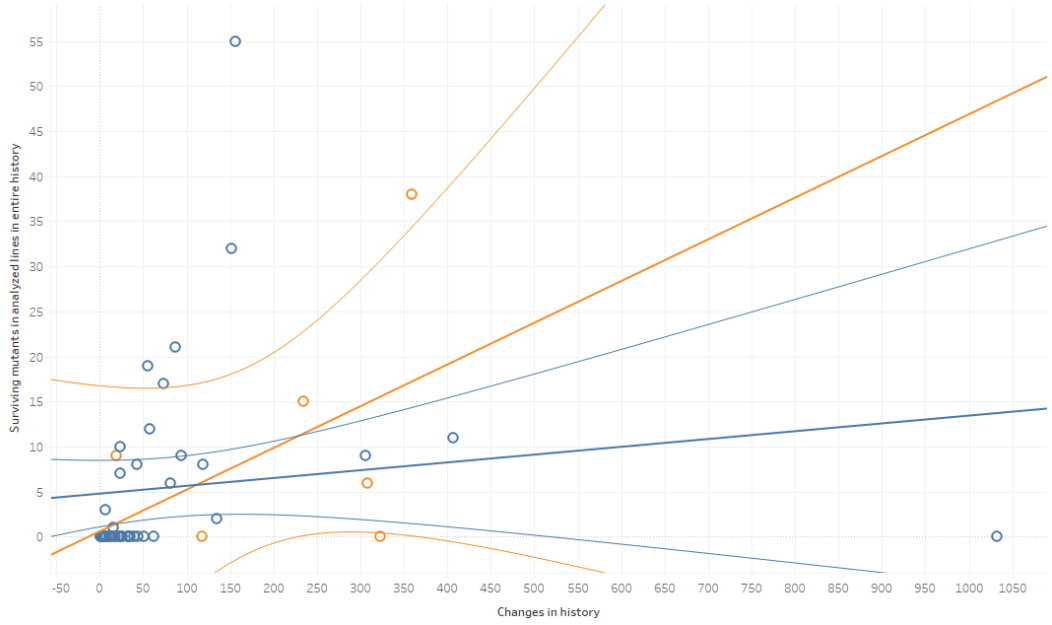


Figure 5.17: Commons Compress Churn vs Survived Mutants computed with OPi+ for each class(p=0.160202), highlighted with orange Bad Smell Classes

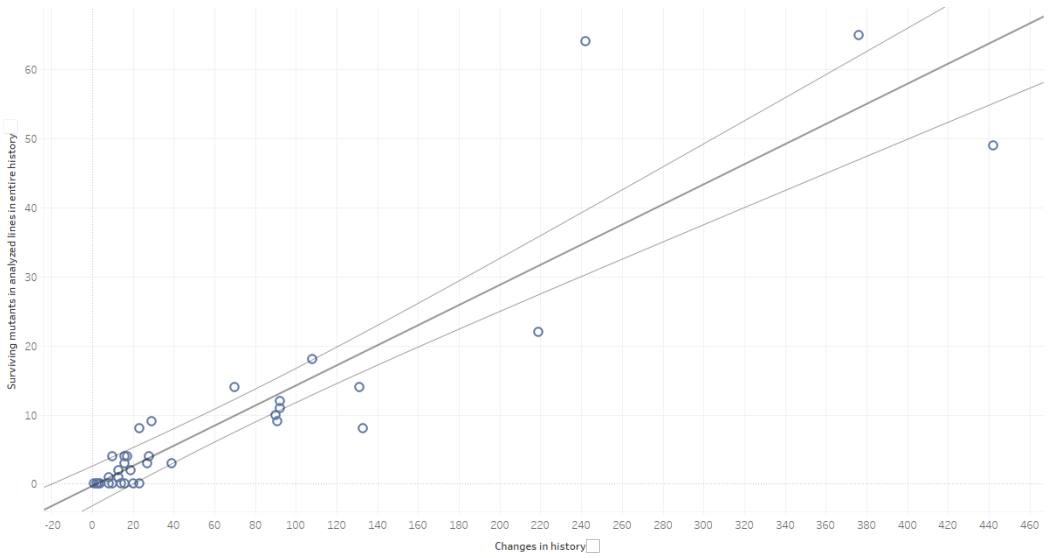


Figure 5.18: Commons IO Churn vs Survived Mutants computed with OPi+ for each class($p<0.0001$)

5.4.3 Bad smells

Bad smells in the code are base are indicators of a potential problem. These types of problem increase the technical debt and are prime candidates for refactoring. We found that surviving mutants can point to code that should be refactored, as described in Section 5.3. Therefore we would expect to have more mutants in the classes that contain bad smells.

The systems analyzed are very well maintained by an active community. There are very few classes that have symptoms of bad smells. The only bad smells we could find over the entire code base are:

- God Class [64]: is a very complex class that contains functionality that should be distributed in several different classes. They are all-knowing classes, communicating with less complex classes.
- Feature Envy [35]: is a method that uses data from a different class to make its computations. This means the method should most likely be in the other class.
- Data Clump [35]: is a set of objects that always show up together. They should have their own object that contains them so that only one object reference is being passed around.

We plot the positioning of the bad smell classes in contrasts with the positioning of all classes based on surviving mutants x-axis and churn y-axis. This is done for both commit history, OPi+, and last version of the files, Pitest. Nevertheless, we have to few data to infer any correlation.

- **JSoup** only has one bad smell class, highlighted in Figure 5.16.
- **Commons Compress** has 5 God classes [64], 2 Feature Envy [35] and one Data Clump [35], highlighted in Figure 5.17.
- **Commons IO** has no bad smell classes.

5.5 RQ5: Can continuous mutation testing be used in the code review process?

We analyzed the latest 5 pull requests (PR) from JSoup, since this is the most active out of the 3 systems analyzed. By analyzing a PR with OPi+ we basically look at each commit from the PR through the continuous mutation testing perspective. We used the same approach we did when we analyzed an entire system commit history. This approach is described in Sections 4.3.1 and 4.3.2. Based on the PR content and OPi+ insight we answer these questions for each of the pull requests:

1. What is the branch coverage?
2. Which mutants were killed?
3. Did OPi+ find code that should be better tested? (Type-5)
4. Why was the code not tested? (not important enough to require tests, too simple or too complicated)

The results are as follows:
PR862: Speed up matching CSS selectors and other tasks which use elementSiblingIndex()
The branch coverage at the beginning of the PR is 83.3% and drops to 82.8% after all commits from the PR are merged. After performing continuous mutation testing analysis on the commit set from the PR, we have 25 killed mutants, out of which the most popular mutator is member variable with 9 killed mutants. The distribution for the line types is as follows:

- Type-1: 3
- Type-2/3: 48
- Type-4: 55
- Type-5: 4

We found 4 instances of Type-5, which require more thorough tests.

```
1 private Map<Element, Integer>
      childElementToSiblingElementIndex = null
      ;
2 if(this.childElementToSiblingElementIndex ==
       null) {
3 this.childElementToSiblingElementIndex =
      null;
4 childNodes = new ChangeNotifyingList<Node>(
      new ArrayList<Node>(4),
      getOnChildNodeChangeRunnable());
```

Three of them were variable initialization not tested and the other one required more tests for the Element class. The missing test relates to the method that is supposed to return the child's position in element sibling list. The test class for Element has on average 5 asserts per test case. Several test cases do not offer full branch coverage. It may be that the sibling list positioning is due to an overall overlooking of the Element class.

PR849: Improved node traversal (including skipping of subtrees, filtering)

The branch coverage at the beginning of the PR is 82.5% and increases to 82.6% after all commits from the PR are merged. After performing continuous mutation testing analysis on the commit set from the PR, we have 0 killed mutants. This is due to the fact that this pull request contained 3 unparsable commits by Cobertura. This leads to very little data to mutate and analyze. The distribution for the line types is as follows:

- Type-1: 9
- Type-2/3: 4
- Type-4: 0
- Type-5: 0

PR815: Fix for Issue#815, taking lowercase key in each case for Attribute and BooleanAttribute

The branch coverage at the beginning of the PR is 82.2% and increases to 82.5% after all commits from the PR are merged. After performing continuous mutation testing analysis on the commit set from the PR, we have 1 killed mutant. This is due to the fact that Pitest crashed for two of the commits. The distribution for the line types is as follows:

- Type-1: 4
- Type-2/3: 0
- Type-4: 1
- Type-5: 1

The Type-5 line is related to an untested variable initialization.

```
1 this.key = key.trim();
```

Based on our research there are many untested variable initializations. However in this case, the value comes from the Java lang package. Nevertheless the initialization should still be tested in the context of the system.

PR717: Text within an element to be queried without doing normalization as is

The branch coverage at the beginning of the PR is 83.1% and remains the same after all commits from the PR are merged. After performing continuous mutation testing analysis on the commit set from the PR, we have 30 killed mutants. The most popular mutator is remove conditional, with 18 killed mutants. The distribution for the line types is as follows:

- Type-1: 0
- Type-2/3: 4
- Type-4: 12
- Type-5: 1

This PR was the best in terms of continuous mutation testing because most of the lines are Type-4 and a few of them are Type-23, incompatible with the operator set, with no Type-1 lines. There was also one Type-5 line caused by an untested variable initialization.

PR384: Add Prototype Style DOM Navigation

The branch coverage at the beginning of the PR is 83.8% and remains the same after all commits from the PR are merged. After performing continuous mutation testing analysis on the commit set from the PR, we have 0 killed mutants. This is due to the fact that Pitest crashes for one commit. The distribution for the line types is as follows:

- Type-1: 65
- Type-2/3: 18
- Type-4: 4
- Type-5: 0

In this case, continuous mutation testing does not discover any special cases. Most is code is simply not tested.

Chapter 6

Discussion

6.1 Result interpretation

Based on the data collected via OPi+, which we presented in Chapter 5, we draw the following conclusions.

6.1.1 RQ1: How is the mutant generation, time and developer effort of mutation testing impacted in a continuous integration environment?

All resources required for mutation testing are reduced in the context of continuous mutation testing. The **number of generated mutants** is reduced in all aspects. We have an 80% reduction in classes analyzed, 91% reduction in mutants generated, with a 95% reduction in generated surviving mutants. We can normalize the number of required mutants by dividing them to the number of classes from the system(NOC). This gives us an absolute number of 166 mutants required for a class when analyzing the entire system. When analyzing the system in a continuous mutation testing manner we only require an average of 20 mutants for a class. For surviving mutants, we require 56 mutants for a class when analyzing the entire system and 3 per class for the continuous approach. The most important outcome of this data is that the absolute number of mutants is manageable whereas the system level numbers are not.

Also, OPi+ requires fewer surviving mutants than Pitest, as shown by the difference between the x-axis and y-axis in Figures 5.4–5.6. This is because by looking at changed code, continuous mutation testing automatically ignores unused code.

By filtering out unused code, OPi+ filters out unmodified code part that include dead code. This has positive outcomes, since not analyzing dead code means there are no dead code equivalent mutants generated. The main cause of equivalent mutants is mutating unused code [41]. Identifying equivalent mutants is a non-decidable problem, being one of the main issues mutation testing is facing [41]. Nevertheless being able to not even generate them is a very important improvement for mutation testing. We believe the smaller number of generated surviving mutants is due to the fact that equivalent mutants were avoided, since previous empirical studies showed that 10 to 40 percent of mutants are equivalent [56].

In our manual analysis, described in Section 5.1.3, we could not identify any equivalent mutant from the surviving mutant pool. The equivalent identification should be made by a system expert.

The amount of analyzed lines has a direct impact on the **manual analysis** required for this process. As shown by the results from the empirical study, described in Section 5.1.1, OPi+ analyzes less surviving mutants than Pitest. When comparing the number of lines analyzed by OPi+ and Pitest per file we have a 95% reduction. More than this we have also a 95% reduction in the number of surviving mutants generated by OPi+ and Pitest per file. This is a direct consequence of the fact that OPi+ generates a significantly smaller number of surviving mutants for large files. All files situated on the right side of the plot, in Figures 5.4–5.6, represent big files that were not changed very often but Pitest generates a lot of surviving mutants for them. Avoiding manual analysis for most of the mutants generated with Pitest, is the main factor that streamlines the continuous mutation testing process. In the same Figure, we notice that we only have one class situated in the top right corner, in all 3 systems analyzed, meaning this is the only case per system in which OPi+ generated a similar number of mutants as Pitest.

The **time of running mutation testing in a continuous environment** is also highly reduced as shown by the data collected, since analyzing a few changed classes takes a lot less than analyzing the entire system. This data is presented in Section 5.1.2 We can notice that the time for analyzing a commit is below the 10 minute threshold [52] in the context of the median for build time being 8.3 minutes [22]. More than this, in our prototype tool we only trigger the actual mutation testing step on selected commits. Nevertheless the steps that lead us to infer the triggering are already custom steps in the build process(testing, diff analysis and code coverage). This means that the continuous mutation testing approach requires the actual mutation testing phase on only selected commits and does not influence the build time of the remaining commits.

It needs to be considered that OPi+ does some extra analysis apart from running Pitest on a subset of classes which adds some time. Also, considering the fact that OPi+ was built on an infrastructure incompatible with continuous mutation testing, the time results show very positive results.

The **time for analysis** reported in Chapter 5 does not represent the time it will take for a real continuous mutation testing pipeline to run, but it is a maximum limit which no real running time will surpass. In our case, the following limitations of Pitest and Operias increase the time OPi+ takes to analyze the commits. In a real environment, where these limitations do not exist, the time will definitely be smaller.

- Pitest analyzes more code than required by continuous mutation testing process

- The Operias algorithm for detecting the code change is being duplicated since the code change is already recorded in the control version system

6.1.2 RQ2: What are the implications of continuous mutation testing on the completeness of currently available operator sets?

On average, 45% of the changed code lines could not be mutated because of the lack of operators. This fact shows a need for additional operators. The high percentage needs to be considered in the context of the fact that all systems analyzed are object-oriented. Even though Pitest operators can be applied on these systems, they are not custom operators for object-oriented features. There are mutation testing papers that propose object-oriented mutation operator sets [41]. However, these operators were inspired by object-oriented features not by real code for which no other operators were found.

Based on the most modified unanalysable lines, we propose an additional set of object-oriented mutators, found in literature, to be implemented with priority. We describe the object-oriented mutators selected from literature in Appendix C. The selection of the terms is described in Section 5.2. We classify the terms computed from Type-2 lines via TFxIDF, from Table 5.10, in 7 categories and suggest an appropriate mutator:

1. **object-oriented modifiers** (e.g. public, private, final, static) - AMC, JSI, JSD [48] described in Appendix C. The object-oriented operators from literature would address the modifiers. However, a valid mutator must pass the compilation phase. This means the only valid mutations left regarding the modifiers would be increasing the visibility scope (e.g. from private to public) or editing the static feature. Even though these operators address the modifiers, they do not introduce relevant faults.

   Nevertheless, the fact that the modifiers are in the top of our TFxIDF analysis means that the lines that contain modifiers are frequently not mutated in the code base. Our research does not necessarily point to mutating the modifiers but rather points to mutating class attribute declarations. To this end we propose a new operator.

   These modifiers most likely are part of a class attribute declaration that is not initialized. If there would be an initialization(e.g. public int x=10;), the already available Inline Constant Mutator from Pitest would have changed the constant value. Since the variables are not initialized, there is no compatible mutation. Nevertheless these lines should be mutated. This can be done through a similar operator that would initialize variables at declaration time. Bad initialization values is a realistic fault that can stay in the system if the constructors are not properly tested.

RDI = Reference Declaration Initialization
Adds an initial value for a class attribute at the reference declaration time.

Original Code	**Mutant**
private int x;	private int x = 10;

2. **data types** (e.g. boolean, int): IHD, IHI [48] described in Appendix C. Here we suggest the same previous new operator.

3. **object-oriented features** (e.g. new, this): PMD, PNC, JTI, JTD [48] described in Appendix C.

4. **values** (e.g. true, false): current Pitest Inline Constant Mutator should be improved to cover all cases.

5. **system specific data structures** (e.g. node, archivers, zipfile, xmlstreamreader): PMD, PNC [48] described in Appendix C.

6. **system specific popular methods** (e.g. add, update): MIR [42] described in Appendix C.

7. **return statement**: FAR [42] described in Appendix C.

We also recorded the type of mutant generated for each system for each size commit cluster, presented in Figures 5.7–5.9. We can notice that regardless of the size of the commit we have the same distribution of generated mutants. Jsoup is more prone to Remove Conditionals Pitest mutator, whereas Commons Compress and Commons IO is also prone to Inline Constant mutator. This distribution difference indicates there is a correlation between nature of the code base and the generated mutants. Due to this correlation and the fact that each system had different popular untested terms, a system could benefit from a few custom made mutators.

6.1.3 RQ3: What feedback information do the surviving mutants offer to developers in the context of a continuous integration setting?

By manual analysis, we compared the current versions of the systems with old ones in order to showcase the evolution of surviving mutants. There are 3 possible situations a mutant can evolve into: deletion of the line for which the mutant was generated, changed or the same mutant status. The distribution is presented in Figures 5.10–5.12. The data is described in Section 5.3

We noticed that 79% of the cases where a line containing a surviving mutant was deleted was due to refactoring. This shows that present surviving mutants might point to code that will be refactored in the future.

A change in the mutant status means a decrease or increase in the number of surviving mutant which was caused by a change in the test suite. Even though the systems we analyzed have high coverage, we found a few cases where tests were deleted carelessly which led to an increase in the number of surviving mutants. The ability of detecting these cases is useful in the code review process in order to avoid deleting important tests. Test suites should be cleaned but not by producing untested code. A decrease in the number of surviving mutants meant the new test cases were added later, when they should have been added earlier. This could have happened because of more time and resources allocated later. This shows that detecting the area that will require testing at the time of the commit could turn out to reduce the resources required over the entire development process.

The rest of the surviving mutants selected by OPi+ pointed to missing test cases. However, not all these cases are relevant for the system's functionality. Almost half of them pointed to variables that had no tests for the initialization value. We manually analyzed the other cases and discovered some potentially useful missing test cases. Nevertheless, the

real value of these cases must be analyzed by a system expert. Each case is described in Appendix B. Also the impact of the missing test case is part of a mutation testing study which is partially outside the scope of this study.

6.1.4 RQ4: Is there a correlation between surviving mutants and system characteristics such as: code coverage, churn and bad smells?

If code coverage would be an accurate indicator of the test suite quality we would expect files with high coverage to have less surviving mutants. However, our data, presented in Section 5.4.1, shows this is not the case. We could not infer any correlation between branch coverage and surviving mutants selected by OPi+. In our study files that are not tested at all generate no coverage mutants. No coverage mutants are ignored and not considered survived mutants within this study. This disassociation is more visible in the branch coverage bin plots, Figures 5.13–5.15.

Based on all the data collected we could infer a correlation between churn and survived mutants selected by OPi+. Data is presented in Section 5.4.2 When we plot the number of lines analyzed by Pitest and OPi+ we see the influence of the change history. This shows that the number of survived mutants that have to be analyzed with the OPi+ approach is related to the change history.

We could not infer any reciprocity between continuous mutation testing and bad smells. This is due to the fact that the analyzed systems had very few bad classes. The data collected is presented in Section 5.4.3. The one class from JSoup that has bad smells was changed less than 50 times therefore has no surviving mutants selected by OPi+, see Figure 5.16. Common Compress has 8 bad smell classes, 3 of which have surviving mutants selected by OPi+ and were changed more than 100 times, see Figure 5.17. Out of these only one class refactored the lines that contained survived mutants. OPi+ focuses on classes that have a high churn. It is possible that classes with bad smells are less edited, thus being ignored by OPi+. Further research is needed in this area.

Nevertheless, the fact that we have a correlation between the number of changes per file and the number of surviving mutants selected by OPi+ per file, supports the approach of change driven mutation testing.

6.1.5 RQ5: Can continuous mutation testing be used in the code review process?

As shown by our JSoup pull request analysis, described in Section 5.5, the feedback from continuous mutation testing describes the quality of the tests that cover the changed area. Not only can it provide an update on the mutation score, but it can also identify specific lines that are not tested well enough. Therefore OPi+ can be used to assess mutation score improvement just like Operias provides updates on the code coverage evolution.

6.2 Threats to Validity

The main threats to validity are caused by one of the following facts:

1. not all commits could be analyzed
2. the limitations of the systems analyzed
3. inaccurate timing
4. limitation of the available operator set

We were **not able to analyze all commits**. Nevertheless, the crash threshold is quite low, and we don‘t believe those commits would have changed the outcome of this study. We recorded all situations where the commit could not be analyzed and clustered them based on their root cause. An overview of the distribution of crashes is given in Table 6.1, as described in Section 4.3.1.

Continuous mutation testing checks every commit that contains at least one change in the code base. Therefore, one type of commit that could not be analyzed is the one that contains **no change of the code base**. Detecting these commits is done in the prefiltering step, which was improved using the first system analyzed. To ensure the integrity of the empirical study, we did not change the prefiltering for the following systems that were analyzed. In Table 6.1 we show how many of the total commits from these systems passed the prefiltering step even though they should have been filtered out.

		JSoup			Commons Compress			Commons IO		
Commits Analyzed		**172**	**113**	**400**	**184**	**327**	**805**	**38**	**47**	**328**
Pitest	**Build Failed**	3.50%	2.60%	2.20%	0.54%					
	No change detected	7.60%	6.19%	2.70%				2.63%		0.61%
	Crash- exit required	6.40%	3.50%	3.75%	0.54%	0.61%	52.30%	2.63%		
Operias	**Cobertura Thread Crashed**				3.80%	2.14%	0.50%		4.26%	1.22%
	Timeout				5.43%	7.95%	3.11%			
OPi+	**Should have been Prefiltered**	2.30%	2.60%	13%	16.30%	14.68%	9.94%	44.74%	42.55%	32.32%
Incompatible	**No pom File**	0.90%			0.54%					
	Unstable Version				38.59%	44.34%	20.37%	5.26%	12.77%	10.06%
Total Failures/Commit Size Category		**20.70%**	**14.89%**	**21.65%**	**65.76%**	**69.72%**	**86.21%**	**55.26%**	**59.57%**	**44.21%**

Table 6.1: Unanalysable Commits Overview

Also, the **analyzable systems space was narrowed** by the constraints imposed by the systems used in OPi+. Both Pitest and Operias are compatible with Maven projects, so we only targeted our system selection procedure toward Maven projects. Nevertheless, not all the selected projects were Maven projects from their beginning or had always a stable state (where all tests pass). This added a temporal limitation on the analysis of this kind of projects, since we could not analyze their entire history. An overview of the incompatible commit distribution is given in Table 6.1.

6.3 Future Work

Continuous mutation testing study can improve in one of the following ways:

1. further develop OPi+ from prototype to product (improve compatibility, performance, user interface)
2. research if prioritization of the lines of code could streamline the process
3. use a different way for evaluation
4. research the impact of continuous mutation testing on other development practices

OPi+ was built as a prototype to research continuous mutation testing. Even though this tool is compatible with popular frameworks such as Java Maven projects the **compatibility** should be extended to other languages and platforms. The first prefiltering step, that identifies whether actual code was changed, must be developed for more languages. The compatibility can also extend to the type of repository used. Nevertheless, the current modular architecture of OPi+ can easily accommodate the interfaces required for all these upgrades.

At the same time, the **performance** of this tool can be improved by parsing information straight from GitHub. The information of the actual change is currently being duplicated by the Operias diff algorithm. Operias is used by OPi+ for getting branch coverage information. In order to avoid code duplication we propose two solutions. The first proposal is that Cobertura can be run as an independent step. The second solution can be incorporating branch coverage analysis in Pitest.

In order to upgrade a prototype to a product, a **user interface** is also required. This should contain new reports and GitHub notifications. Since OPi+ offers a more thorough analysis than Pitest, the reports must be upgraded to include only the data considered important by OPi+. More than this, based on our preliminary results, continuous mutation testing can also be used in analyzing the quality of a pull request, thus becoming part of the code review process. Even though the connection to the GitHub API is already implemented via Operias, the final reports and code review information have to be linked back to the repository.

Due to the time limitation of this study we were not able to research the prioritization of the lines of code in the context of continuous mutation testing. Currently OPi+ does not prioritize the Type-5 lines. This could be done based on the type of change committed. Fluri et al. [30] implemented an **automatic change classifier**, ChangeDistiller. This is an Eclipse plugin that classifies changes and rates their importance for Java systems. It uses an algorithm (described in [31]) that detects changes between two versions of the system, based on tree edit operations (insert, delete, move,or update of tree nodes). The authors, described all the types of changes detected by the algorithm and classified it as changes that modify or preserve functionality which we summarize in Table 6.2. Also, they gave to each of these type an importance rating (low, medium or high). In our study, only functionality-modifying changes are within the scope of mutation testing since functionality-preserving changes will result in equivalent mutants. Table 6.2 highlights in blue all changes that are functionality-modifying and have a crucial impact. Since each type of change has an operator that can be applied to it, we can prioritize the operators based on the importance rating of the type of change from this table. Prioritizing the mutants based on potential impact can

maximize the potential of mutation testing, even if the process can not be finished due to time constraints.

The purpose of this study was to research the potential of continuous mutation testing. The value of mutation testing in itself is an independent study conducted by several researchers [41]. Apart from the method used in our study, another **good way to evaluate** continuous mutation testing would be to apply this process on a system that has previously been analyzed to showcase the benefits of mutation testing. This way, we can check if all Type-5 lines detected by OPi+ contain the most important mutants discovered by mutation testing. This would prove that continuous mutation testing is a viable solution because it produces faster the same results as applying mutation testing in the way it is currently applied. In our current study, the evaluation of Type-5 mutants lacks a system expert that can accurately evaluate their impact on the system. To our knowledge, there are few papers concerning the impact of surviving mutants (e.g. [70]). The projects used by [70] are 18 popular short algorithms written in C, such as Bubble Sort, Min, Prime numbers etc. For an evaluation like ours, we would require a Java Maven project, developed for production, with an open source history analyzed by a mutation testing paper.

Continuous mutation testing (CMT) is a software development practice because it is applied at every change of code. This is why, if used, it will impact the entire development process. We would recommend a real time monitoring of how this practice would **impact** the team and other software development practices.

Table 6.2: Types of change as described by Fluri et al. [30]. Blue highlighted changes are functionality modifying and crucial impact. This makes them prime candidates for important mutation testing operators

Category	Type	Functionality	Impact
Method body changes	STATEMENT ORDERING CHANGE	modifying	low
	STATEMENT PARENT CHANGE	modifying	medium
	STATEMENT INSERT	modifying	medium
	STATEMENT DELETE	modifying	medium
	STATEMENT UPDATE	modifying	low
Structure statements	Increasing statement insert	modifying	high
	Decreasing statement delete	modifying	high
	Condition expression change	modifying	medium
	else part insert	modifying	medium
	else part delete	modifying	medium
Class body changes	Additional object state	preserving	low
	Removed object state	modifying	crucial
	Additional functionality	preserving	low
	Removed functionality	modifying	crucial
Access modifier changes	Increasing Accessibility Change	modifying	medium
	Decreasing Accessibility Change	modifying	crucial
Final modifier changes	Final Modifier Insert	modifying	crucial
	Final Modifier Delete	preserving	low
Attribute declaration changes	Attribute Type Change	modifying	crucial
	Attribute Renaming	preserving	high
Method declaration changes	RETURN TYPE INSERT	modifying	crucial
	RETURN TYPE DELETE	modifying	crucial
	RETURN TYPE UPDATE	modifying	crucial
	METHOD RENAMING	preserving	high
	PARAMETER INSERT	modifying	crucial
	PARAMETER DELETE	modifying	crucial
	PARAMETER ORDERING CHANGE	modifying	crucial
	PARAMETER TYPE CHANGE	modifying	crucial
	PARAMETER RENAMING	preserving	medium
Class declaration changes	CLASS RENAMING	preserving	high
	PARENT CLASS INSERT	modifying	crucial
	PARENT CLASS DELETE	modifying	crucial
	PARENT CLASS UPDATE	modifying	crucial

Chapter 7

Related Work

7.1 Improving Continuous Integration

Even though the continuous integration approach improved the software development process, a high number of failing builds in the history of a project can be a threat to the efficiency of the approach by Leitner et al. [63]. Through empirical study [63] the authors found that more than 80% of build failures are caused by failing integration tests as shown in Figure 7.1. They also discovered that the strongest influencing factor for failing builds is the stability of the build system in recent history. The impact tests have on the build process shows a need for ensuring their correct functionality.

In order to avoid failing tests, projects use code analyzing tools that are especially cre-

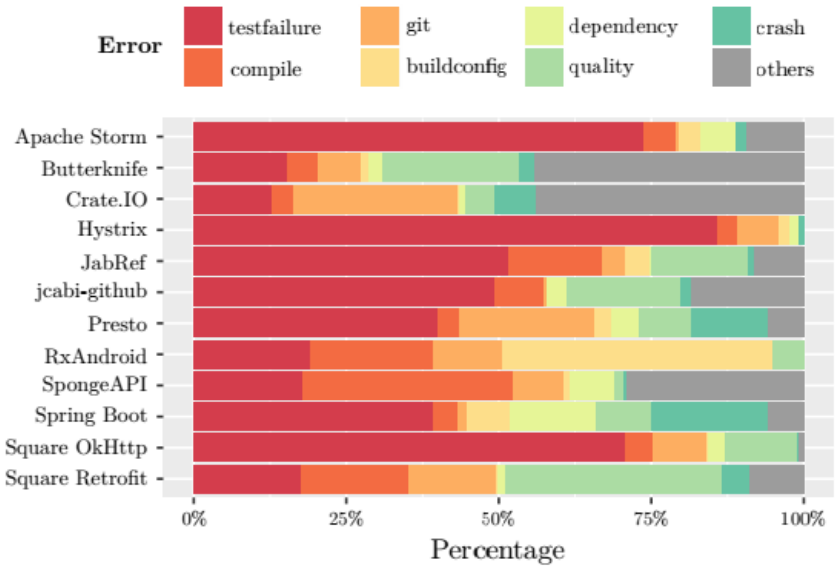


Figure 7.1: Distribution of common build error categories [63]

ated to pair with the continuous integration pipeline. For example, Infer[1] is a tool that looks for bugs such as null pointer references, resource and memory leaks and concurrency issues. It is a static code analyzer that does not change the code base or inject faults, but it achieves additional code checking, independent of the test suite, in a continuous integration setting.

Aside from static analysis, "Software testing is well established as an essential part of the software development process and as a quality assurance technique widely used in industry." [23]. However, maintaining an effective test suite is a problem most projects struggle with. There are best practices for how to write good quality code but only a few guidelines on how to write good tests [51]. This creates a need for a more thorough way of analyzing the effectiveness of a test suite.

7.2 Techniques to Optimize Mutation Testing Usability

A solution for testing the effectiveness of a test suite is mutation testing, presented in Chapter 2. A recent study by Zhu et al. [62] from 2016 presents the practical usage of mutation testing as described by other studies. The researchers based their findings on 159 papers making this survey very thorough. More than half of these papers do not mention how they deal with one of the biggest problem mutation testing faces, labeling the seemingly similar faulty versions. The rest of the papers either ignored those cases or labeled them by manual analysis. Even though most papers do not mention how they dealt with overcoming the challenges that come with mutation testing there are some studies that exclusively proposed solutions to improve mutation testing. Other studies even developed an infrastructure to make mutation testing easy to use. These studies are presented in the following sections.

7.2.1 Cost Reduction Techniques

The mutation testing process requires significant amount of computational resources. Therefore, efforts have been made to reduce this cost. The cost reduction techniques developed so far focus on one of two directions: 1) reducing the number of generated mutants or 2) reducing the overall costs required by the process [41].

Methods proposed for **reducing the number of generated mutants** focus on finding the subset of all generated mutants that would output a similar mutation score. The way the mutants that belong to that subset are chosen, is so far computed using four different approaches [41].

Methods proposed for **reducing the execution cost** are focused on optimizing the mutant execution process. This can be done through runtime optimization techniques [27] which implement the mutants at bytecode level thus canceling the compilation process. Another way of reducing the cost is the concurrent execution of mutants on different machines [45]. A final proposed solution is weak mutation. This approach is checking intermediate states of the system. This way if an intermediate state is not correct the mutant can already be considered killed. This way the killed mutants can be identified faster and execution time is shorted since not all states must be computed [69].

[1] http://fbinfer.com/docs/about-Infer.html

Mutation testing is a very expensive approach also because of the need for manually labeling the surviving mutants. There are also a few techniques proposed to streamline this process [21, 21, 16, 29, 37, 60, 66]. One solution is to detect the equivalent generated mutants by optimizing the code [21] or by analyzing the input constraints [67]. Automatically detecting the equivalent mutants can be done through the co-evolutionary approach [16] which uses a fitness function designed to have poor value for the equivalent mutants. Another way is to consider that equivalent mutants are the ones with the same output but different execution time or memory usage [29], or those that have the lowest impact on coverage change [37]. The code coverage is computed by looking at the bytecode [60], however this method does not apply to mutants that are already implemented in the bytecode. Another proposed solution is called Infection Analysis [66] and it is based on the probability of a mutant changing a data state.

7.2.2 Infrastructure

Even though the cost reduction techniques described in the previous section streamline mutation testing, current mutation testing tools do not focus on them [62]. The lack of interest in identifying equivalent mutants is not only observed in research papers but also in the available features of mutation testing tools. Current mutation testing tools provide basic equivalent mutant detection techniques in the best cases. As described by Zhu et al. [62], only 1 in 7 tools for Java provide an equivalent mutant detector feature. None of the 2 tools presented by Zhu et al. [62] for C# have this feature. For other non object-oriented languages that make use of mutation testing, out of the 10 tools, Zhu et al. [62] found only 3 tools have this option.

The most well maintained mutation testing tool, Pitest, does implement several cost reduction techniques but the time required to run Pitest and manually sort the useful information from the Pitest reports is longer than the time limit required by a CI feedback loop. In a continuous integration environment the average for receiving feedback for a change is less than the time required for running mutation testing in its current state (e.g. Facebook feedback rule is 10 minutes [52]).

The current OPi+ mutation black-box makes use of Pitest, one of the most popular and compatible mutation testing tools. Just as all the other mutation testing tools, Pitest was designed to be run on the entire system. However the Pitest community contributed with a feature that allows to run Pitest on all the files that were changed in the last commit [15].

In principle Pitests compatibility with Maven facilitates a CI compatible infrastructure. Unfortunately, in order for mutation testing to be usable in a continuous integration environment, more steps need to be added because of the following problems: Pitest analyzes more than required, it provides no filtering nor prioritization of cases, it does not help in any way with the manual analysis still required after the mutation testing is done. Also, the reports Pitest generates are very well structured, but since they contain all the information regarding the mutation process, the useful information is hard to find.

7.2.3 Operator Set Improvement

The current operator set of Pitest is proven to be feasible and practical on real world programs [26]. Nevertheless, the community recognizes a need for additional operators. One example is the Pitest-Descartes research conducted as part of the Stamp project [5]. They base the need for additional operators on a study conducted to evaluate the validity of code coverage as a measure of test effectiveness [54]. However, to our knowledge, our study is the only research that proposes a set of new operators based on high impact code that is incompatible with the current operator set.

Chapter 8

Conclusions

”I vividly remember one of my first sightings of a large software project... My manager, part of the QA group, gave me a tour of a site and we entered a huge depressing warehouse stacked full with cubes. I was told that this project had been in development for a couple of years and was currently integrating, and had been integrating for several months. My guide told me that nobody really knew how long it would take to finish integrating. From this I learned a common story of software projects: integration is a long and unpredictable process. But this needn’t be the way.”, Martin Fowler [33].

The way to solve integration problems is by integrating continuously. The practice of continuous integration (CI) implies that developers integrate at least daily [33]. Each change is processed by the CI pipeline, which decides whether the change should be included in the master trunk version of the system. Once the change is merged, the new version of the system is turned into an executable artifact. This process, the CI pipeline, traditionally contains steps for compiling and linking the change. This new version of the system may be able to run, but that does not mean it does the right thing [39].

Within this context, ”Testing has moved to a front and central part of programming.” [51]. This is due to the fact that automated testing is a ”good way to catch bugs” [33]. Practicing automated tests is a ”huge opportunity to programming teams” [51] since it enables ”teams to make drastic changes to a code-base with far less risk...but with these opportunities come new problems and new techniques.” [51]. In the event of a change breaking the system, having a test case fail, the entire build process should fail, thus rejecting the change. Therefore, the test suite ensures the correct functionality of a continuously integrated software system [39].

Since the system’s quality is described by the tests’ outcome, there is a need for a good quality test suite. Currently, in order to assess the quality of a test suite, the code coverage is used as a metric [36]. This approach is very basic since it only checks whether the code is being tested, but not how it is tested.

A method that checks the fault finding capability of a test suite is mutation testing. This approach is based on seeding faults in the system and then checking whether the tests can find the known fault [41]. Therefore, the goal of this thesis has been to explore how mutation testing can be applied to changes under analysis in a continuous integration setting. No other study paired mutation testing with CI, so there is no current infrastructure for a

CI environment that includes the mutation testing process. For our study, **we developed OPi+**, a prototype tool for experimenting the infrastructure was required for a continuous mutation testing approach. Using real-world systems for analysis, **we give initial evidence of the usefulness of continuous mutation testing in terms of costs and benefits when applied to realistic software changes**.

Through setting up a prototype infrastructure, we were able to discover specific outcomes of mutation testing within the continuous integration environment. **We defined 5 types of outcomes** together with a **continuous mutation testing behavior flow**, both described in Chapter 3. The OPi+ architecture combines current technologies for code reviewing(Operias), control versioning(Github), optimized mutation testing(Pitest) and **our own implementation of analysis that streamlines the manual analysis phase** of the mutation testing process. This analysis contains prefiltering and line analysis based on a more in-depth code coverage analysis combined with mutant output. Due to the well setup filters, we reduce the code base analyzed which reduces the number of generated mutants, which also reduces the scope of the manual analysis. The filters **select only impactful code changes**, removing isolated parts of code that are the main source of equivalent mutants and trivial missing test cases.

We then used the continuous mutation testing prototype infrastructure to perform experiments in order to infer the potential of this method on 3 popular, open source, highly tested Java Maven systems. **We performed analysis on the entire commit history of the systems**, inferring a specific output for each line from all compatible commits. By further studying the OPi+ labelled lines **we also analyzed the potential impact on technical debt** and **code reviewing process**. The data from the empirical study showed that there is a correlation between churn and the presence of surviving mutants. This fact backs up the proposed way of using mutation testing in continuous integration environment. More than this, we showed that there is no correlation between code coverage or bad smells and the distribution of surviving mutants. However, the entire process can be used within the context of code reviews. We also found specific untested cases that can impact the entire system.

The main disadvantages of mutation testing are the need for a significant amount of computational resources and the fact that manual analysis is required. No other research paper proposed solving the high cost of mutation testing by reducing the scope of mutations to real time changes, which is what happens in our continuous integration environment. Based on our empirical study **we showed not only that mutation testing in a CI environment requires significantly fewer resources but they are also within the limits required by a CI pipeline**. The infrastructure we created keeps the entire process below the 10 minute feedback time limit, having an 80% reduction in analyzed classes, 91% reduction in generated mutants with a 95% reduction in surviving mutants when compared with current usage of mutation testing over the entire system.

Mutation testing cost reduction techniques are very well studied [41]. Nevertheless, by combining them with a change driven focus, we get an overall higher impact. At the same time we cancel out the main disadvantages that make mutation testing unusable. Based on our thorough line analysis within the scope of continuous mutation testing **we were able to also identify relevant code that is currently unmutable**. In order to solve this problem,

we analyzed the unmutable code and **proposed** a set of operators that should streamline the entire process. Most operators are proposed in literature but we also propose a new operator and suggest improvements for existing mutators.

We also **analyzed the evolution of surviving mutants** through the development of the systems. More than half of the lines with surviving mutants where eventually deleted, most of them because of a refactoring process. This leads us to conclude that surviving mutants should either be killed by a new test case or by refactoring the code.

In our study, *we showed initial evidence that mutation testing can successfully be made compatible with a CI environment*. Continuous mutation testing can find specific missing test cases and detect code that should be refactored, both with minimum noise and manual analysis. The results are very positive, therefore we proposed a few ideas that could possibly streamline the entire continuous mutation testing process further.

Integrating your software improvements should not be depressing, instead it should be a confident continuous process based on tests made thorough through continuous mutation testing. ”Its hard enough for software developers to write code that works on their machine. But even when thats done, theres a long journey from there to software thats producing value - since software only produces value when its in production.”, Martin Fowler [32]. With continuous mutation testing the system can be deployed into production with a higher degree of confidence.

Bibliography

[1] Apache maven project. https://maven.apache.org/shared/maven-invoker/usage.html. Accessed: 2017-5-28.

[2] Asm java bytecode framework. http://asm.ow2.org/. Accessed: 2017-7-5.

[3] Bcel byte code engineering library. https://commons.apache.org/proper/commons-bcel/. Accessed: 2017-7-5.

[4] Cobertura. http://cobertura.github.io/cobertura/. Accessed: 2017-5-28.

[5] Descartes mutation engine for pitest. https://github.com/STAMP-project/pitest-descartes. Accessed: 2017-06-20.

[6] Document object model. https://www.w3schools.com/js/js_htmldom.asp. Accessed: 2017-5-28.

[7] Eclemma. http://www.eclemma.org/. Accessed: 2017-5-28.

[8] Gitective. https://github.com/kevinsawicki/gitective. Accessed: 2017-5-28.

[9] Github is how people build software. https://github.com/about. Accessed: 2017-06-20.

[10] Jacoco. http://www.jacoco.org/jacoco/. Accessed: 2017-5-28.

[11] Jgit library. https://eclipse.org/jgit/. Accessed: 2017-5-28.

[12] Jsoup. https://github.com/jhy/jsoup. Accessed: 2017-5-28.

[13] Maven scm plugin. https://maven.apache.org/scm/maven-scm-plugin/usage.html. Accessed: 2017-07-09.

[14] Pitest report issue. https://github.com/hcoles/pitest/issues/336. Accessed: 2017-5-28.

[15] Pitest run on last commit feature. https://github.com/hcoles/pitest/issues/132. Accessed: 2017-5-28.

[16] Konstantinos Adamopoulos, Mark Harman, and Robert M Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and evolutionary computation conference*, pages 1338–1349. Springer, 2004.

[17] Akiko Aizawa. An information-theoretic perspective of tf–idf measures. *Information Processing & Management*, 39(1):45–65, 2003.

[18] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[19] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[20] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[21] Douglas Baldwin and Frederick Sayward. Heuristics for determining equivalence of program mutations. Technical report, Defense Technical Information Center Document, 1979.

[22] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An analysis of Travis CI builds with GitHub. Technical Report No. e1984v1, PeerJ Preprints, 2016.

[23] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[24] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[25] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[26] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452. ACM, 2016.

[27] Richard A DeMillo, Edward W Krauser, and Aditya P Mathur. Compiler-integrated program mutation. In *Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International*, pages 351–356. IEEE, 1991.

[28] John Donne. *Devotions Vpon Emergent Occasions, and Seuerall Steps in My Sicknes: Digested Into I. Meditations Upon Our Humane Condition. 2. Expostuvlations, and Debatements with God. 3. Prayers, Upon the Seuerall Occasions, to Him*. AM.

[29] Michael Ellims, Darrel Ince, and Marian Petre. The Csaw C mutation tool: Initial results. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 185–192. IEEE, 2007.

[30] Beat Fluri and Harald C Gall. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 35–45. IEEE, 2006.

[31] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[32] Martin Fowler. Continuous delivery. `https://martinfowler.com/delivery.html`. Accessed: 2017-06-19.

[33] Martin Fowler. Continuous integration. `https://www.martinfowler.com/articles/continuousIntegration.html`. Accessed: 2017-06-19.

[34] Martin Fowler. Software development in the 21st century: Continuous delivery.

[35] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[36] Andrew Glover. In pursuit of code quality: Dont be fooled by the coverage report. *IBM Developer Works blog post*, pages 1–2, 2006.

[37] Bernhard JM Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 192–199. IEEE, 2009.

[38] William C Hetzel and Bill Hetzel. *The complete guide to software testing*. John Wiley & Sons, Inc., 1991.

[39] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[40] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[41] Yue Jia, Student Member, and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. 37(5):649–678, 2011.

[42] Sun Woo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing Verification and Reliability*, 11(4):207–225, 2001.

[43] Sunwoo Kim, John a Clark, and John a Mcdermid. Class Mutation : Mutation Testing for Object-Oriented Programs. *Net. Object Days*, pages 9–12, 2000.

[44] Daniel Klischies and Konrad Fögen. An analysis of current mutation testing techniques applied to real world examples. *Full-scale Software Engineering/Current Trends in Release Engineering*, page 13, 2016.

[45] Edward W Krauser, Aditya P Mathur, and Vernon Rego. High performance testing on simd machines. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 171–177. IEEE, 1988.

[46] Richard J Lipton, Richard A DeMillo, and FG Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.

[47] RJ Lipton. Fault diagnosis of computer programs. *Student Report, Carnegie Mellon University*, 1971.

[48] Yu-Seung Ma and Jeff Offutt. Description of class mutation mutation operators for java. *Electronics and Telecommunications Research Institute, Korea*, 2005.

[49] Radu Marinescu, George Ganea, and Ioana Verebi. Incode: Continuous quality assessment and improvement. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 274–275. IEEE, 2010.

[50] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[51] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[52] Alexander Mols. Facebook on testing - guest lecture. *Delft University of Technology, Software Testing and Quality Engineering Course*, June 13 2017.

[53] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 342–352. ACM, 2011.

[54] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. Will my tests tell me if i break this code? In *Continuous Software Evolution and Delivery (CSED), IEEE/ACM International Workshop on*, pages 23–29. IEEE, 2016.

[55] A Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[56] A Jefferson Offutt and W Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[57] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[58] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[59] Sebastiaan Oosterwaal, Arie van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. Visualizing code and coverage changes for code review. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1038–1041. ACM, 2016.

[60] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. *Proceedings - International Conference on Software Engineering*, 1:936–946, 2015.

[61] Pascale Philippe Chevalley, Thévenod-Fosse. A mutation analysis tool for java programs. *Journal on Software Tools for Technology Transfer (STTT)*, 2001.

[62] Andy Zaidman Qianqian Zhu, Annibale Panichella. A Systematic Literature Review of How Mutation Testing Supports Test Activities. PeerJ techreport, 2016.

[63] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 345–355. IEEE Press, 2017.

[64] Arthur J Riel. *Object-oriented design heuristics*, volume 335. Addison-Wesley Reading, 1996.

[65] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

[66] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[67] Jeffrey M Voas and Gary McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[68] Elaine J Weyuker. The applicability of program schema results to programs. *International Journal of Parallel Programming*, 8(5):387–403, 1979.

[69] MR Woodward and K Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 152–158. IEEE, 1988.

[70] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, pages 919–930. ACM, 2014.

Appendix A

Manual Analysis

This Appendix describes the manual analysis done on the data collected by OPi+ as described in Chapter 4. We selected all files that were changed more than 100 times and had surviving mutants. Then for each line we record the following: commit ID, code line content, line number, the type of change, number of survived and killed mutants at commit time and in last version of the system. NC stands for Not Covered mutants. In total we manually analyzed 319 changed lines (94 for JSoup, 101 for Commons Compress, 124 for Commons IO).

Based on this data we identify the line evolution label inferred manually. If the line was deleted we record the reason for the deletion in the Deleted column (R for refactored and F for change in functionality). If the number of mutants changed we note the scenario type in the Changed column (I for improved and D for decreased). If the mutant number is the same we differentiate the initialized cases in the Same column(init for initialized). We use ! to mark important missing test case described in Appendix B.

The complete data collected in stored on the Google Drive Repository and can be accessed at the following links: Jsoup[1], Commons Compress[2] and Commons IO[3]. Some of the information recorded for the systems analyzed is found in Tables A.1, A.2 and A.3.

Line	Type	Survived	Killed	*LastVersion*		*Label*		
				S	K	Deleted	Changed	Same
158	UPDATE	2	2	0	4		I	
210	UPDATE	1	0	1	0			!
15	UPDATE	1	1	1	1			init
71	UPDATE	2	2			R		
164	UPDATE	1	5		8		I	
276	UPDATE	1	6	1	6			!
297	UPDATE	1	0	1				init
273	ADD	1	2			R		
308	UPDATE	1	12			R		
321	UPDATE	2	11	19-NC				!
116	UPDATE	3	13			R		

[1] https://docs.google.com/spreadsheets/d/1dPZMEqutZWP6FtOjjxBxmJEkt7Cf2tf6Klcou1hQsCY/edit?usp=sharing
[2] https://docs.google.com/spreadsheets/d/1PpQD7lPHUmcfTxx9M64duwoD8I6EYdVRlf_W2mmtTuo/edit?usp=sharing
[3] https://docs.google.com/spreadsheets/d/1MEdS-HInY3yT7nv85cxgWmYqw6c8s8nm4dTiTFbyncY/edit?usp=sharing

196	UPDATE	2	4			R		
95	ADD	6	0			R		
97	ADD	6	0			R		
56	UPDATE	2	0			R		
24	ADD	1	0			F		
25	ADD	1	0			R		init
26	ADD	1	0					
27	ADD	1	0					
28	ADD	1	0					
29	ADD	1	0					
49	ADD	5	0			F		
224	ADD	5	0			R		
228	ADD	5	0			R		
239	ADD	2	0			R		
240	ADD	2	0					
245	ADD	2	0					
246	ADD	2	0					
251	ADD	2	0					
252	ADD	2	0					
253	ADD	2	0					
258	ADD	2	0					
414	ADD	1	0			R		
55	ADD	2	0		2		I	
56	ADD	1	1			F		
59	UPDATE	2	0			R		
186	ADD	2	2	0	6		I	
21	UPDATE	2	0			R		
24	UPDATE	1	1	1	1			
25	UPDATE	1	1	1	1			
280	UPDATE	2	0			R		
281	UPDATE	2	0			R		
185	ADD	3	0	0	4		I	
463	UPDATE	1	3	0	7		I	
159	ADD	1	2			F		
110	ADD	1	2			R		
160	ADD	1	0	1		R	I	
28	UPDATE	1	0			R		
238	ADD	1	0			R		
21	ADD	1	1	1	1			init
238	UPDATE	1	0			R		
143	ADD	3	2		6 NC		D	
156	ADD	1	2		5 NC		D	
100	UPDATE	3	0		3 NC		D	
1074	UPDATE	2	5			R		
1079	ADD	1	2			R		
1085	ADD	1	0	1	0			init
1086	ADD	1	0	1	0			init
1089	ADD	1	2	1	2			init
1092	ADD	3	6	1	11			!
1109	ADD	1	6	1	6			!
1110	ADD	1	1	1	4			!
1074	UPDATE	2	5	=				
1079	ADD	1	2					
1085	ADD	1	0					

1086	ADD	1	0					
1089	ADD	1	2					
1092	ADD	3	6					
1109	ADD	1	6					
1110	ADD	1	1					
1066	UPDATE	2	5	=				
1071	UPDATE	1	2					
1080	UPDATE	1	2					
1082	UPDATE	3	6					
1097	UPDATE	1	6					
1098	UPDATE	1	1					
1036	UPDATE	2	5			R		
819	UPDATE	1	1	0	2	R		
888	UPDATE	1	7		9		I	
109	ADD	1	5	1	9	R		
368	ADD	3	0			R		
258	UPDATE	1	2		4	R		
277	ADD	1	2	1	2			!
48	ADD	1	0	1	0			init
248	ADD	1	2		3		I	
181	UPDATE	1	2			F		
103	UPDATE	1	2			F		
98	ADD	1	4			F		
181	UPDATE	1	2			F		
94	ADD	1	0			F		
173	UPDATE	1	2			F		
38	ADD	2	0	2				init
55	ADD	3	0	5				!
41	ADD	1	0	2				init

Table A.1: JSoup Line Manual Analysis

Line	Type	Survived	Killed	*LastVersion*		*Label*		
				S	K	Deleted	Changed	Same
244	UPDATE	1	2	0	5		I	
110	ADD	1	0			R		
811	UPDATE	1	1		3		I	
819	UPDATE	1	1		3		I	
833	UPDATE	1	1		3		I	
835	UPDATE	1	1		3		I	
1046	UPDATE	5	0		6		I	
1049	UPDATE	5	0	6				!
812	UPDATE	3	0		4		I	
828	UPDATE	1	0			R		
831	UPDATE	2	4		8		I	
834	UPDATE	1	3	2	3			!
891	ADD	2	0			R		
892	ADD	3	0			R		
894	ADD	3	0			R		
895	ADD	1	0			R		
665	ADD	1	0			R		
585	ADD	1	0					init
304	UPDATE	1	0			R		

326	ADD	2	0		2			!
196	UPDATE	1	1		2		I	
199	UPDATE	1	2		4		I	
221	UPDATE	1	2		4		I	
253	ADD	1	2			R		
257	ADD	1	2			R		
236	ADD	1	0		2		I	
68	UPDATE	1	1	1	1			init
452	ADD	1	4			R		
475	ADD	1	0			R		
487	ADD	1	0			R		
460	ADD	1	1			R		
514	UPDATE	1	0		2		I	
454	UPDATE	1	0	1	2			!
457	UPDATE	1	2	1	4			!
459	UPDATE	1	0	3				!
463	UPDATE	1	0	2				!
465	UPDATE	1	0	2				!
473	ADD	1	8	1	8			!
305	ADD	1	2	2	4			!
258	UPDATE	2	3			R		
133	ADD	2	3	3	2		D	
366	UPDATE	3	4	1	6		I	!
370	UPDATE	5	0	3	2		I	!
375	UPDATE	1	4	1	4			!
376	UPDATE	1	0		2		I	
382	UPDATE	3	1		4		I	
389	ADD	3	0	3				!
390	ADD	3	0	3				!
391	ADD	3	0	3				!
392	ADD	2	0	2				!
394	ADD	4	4	4	4			!
396	ADD	2	0	2				!
410	UPDATE	1	6	1	6			!
65	UPDATE	1	2		3		I	
328	UPDATE	3	3	2	4		I	!
330	UPDATE	2	2			R		
357	UPDATE	1	0	2	1			!
600	ADD	1	0			R		
601	ADD	1	0		1		I	
602	ADD	1	0	1				init
605	ADD	1	5		6		I	
607	ADD	2	8		10		I	
609	ADD	2	3			R		
610	ADD	4	0			R		
611	ADD	1	0			R		
615	ADD	1	2		3		I	
616	ADD	1	0		1		I	
196	ADD	1	2	1	3			init
226	ADD	1	0		1		I	
227	ADD	1	0		1		I	
262	ADD	2	0	2				init
264	ADD	1	0			R		
267	ADD	1	0	2				init

332	ADD	1	0	1				init
285	ADD	1	2		4		I	
177	ADD	1	0		1		I	
183	ADD	1	0	1				init
188	UPDATE	3	0	1	2		I	!
484	ADD	1	2		3		I	
524	ADD	1	1	1	2			init
530	ADD	1	0			F		
545	UPDATE	1	1	1				init
217	UPDATE	2	0	2				init
218	UPDATE	2	0	2				init
545	UPDATE	1	0	1				init
548	UPDATE	4	2	1	5		I	!
79	UPDATE	1	1		2		I	
85	UPDATE	1	1	1	2			init
275	UPDATE	1	0	1	2			init
581	UPDATE	2	1		6		I	
441	UPDATE	4	0					
447	UPDATE	3	0					
509	UPDATE	6	0					
507	ADD	6	0					
769	ADD	1	0					
143	ADD	3	0	2	1			init
144	ADD	2	0	2				init
145	ADD	1	1			F		
146	ADD	2	0	2				init
157	ADD	2	0	2				init
159	ADD	1	2		3		I	

Table A.2: Commons Compress Line Manual Analysis

Line	Type	Survived	Killed	*LastVersion*		*Label*		
				S	K	Deleted	Changed	Same
1452	UPDATE	1	0			R		
1503	ADD	1	0			R		
1646	UPDATE	1	0			R		
1695	ADD	1	0			R		
1483	UPDATE	3	2	3	2			!
1633	UPDATE	3	2	3	2			!
476	UPDATE	1	2	1	2			!
1482	UPDATE	3	2			R		
1631	UPDATE	3	2			R		
1857	UPDATE	1	2	2	1			!
1861	UPDATE	1	2	1	1			!
1896	UPDATE	3	0	1	2		I	
1900	UPDATE	1	1	1	2			!
1926	UPDATE	1	4	1	4			!
1793	UPDATE	1	0	1	2			!
347	ADD	1	2			F		
1489	UPDATE	1	0			R		
1481	ADD	3	2			R		
1487	ADD	1	0			R		
1497	ADD	3	2			R		

1630	ADD	3	2			R		
1636	ADD	1	0			R		
1646	ADD	3	2	3	2			!
426	ADD	1	2	1	4			!
1825	ADD	1	0	1	1			!
1869	ADD	1	0	1	1			!
381	ADD	1	4			F		
405	ADD	1	2			F		
367	ADD	3	3	3	3			!
1727	UPDATE	3	0	4	0			init
1732	UPDATE	4	1	4	1			!
1733	UPDATE	1	0	2	0			!
1736	UPDATE	1	0	4	0			!
2441	ADD	3	3	0	7		I	
2455	ADD	1	2	1	3			!
2361	UPDATE	2	4	0	6		I	
2368	UPDATE	2	0			R		
2369	UPDATE	1	0			R		
1143	UPDATE	1	2	4	1			init
1560	ADD	1	2	1	2			!
1565	ADD	1	2	1	3			!
1595	ADD	1	2	1	2			!
1727	UPDATE	3	0	4	0			init
1709	UPDATE	1	0	1	0			init
1712	UPDATE	4	0			R		
1713	UPDATE	1	0			R		
1714	UPDATE	2	0			R		
1715	UPDATE	1	0	2	0			!
1719	ADD	1	0	4	0			!
1727	UPDATE	1	2	1	2			!
136	ADD	1	2	1	2			!
154	ADD	1	2	1	2			!
159	ADD	1	2	1	2			!
328	UPDATE	4	2	4	4			!
2380	UPDATE	1	2	2	2			!
2108	ADD	1	0	1	1			!
223	ADD	1	0	1	2			!
1498	ADD	1	0	1	1			!
1531	UPDATE	1	0	1	1			!
1585	ADD	1	0	1	1			!
1617	ADD	1	0	1	1			!
1656	UPDATE	1	0	1	1			!
1725	ADD	1	0	1	1			!
1362	UPDATE	1	2	1	3			!
1385	ADD	1	1	1	2			!
1391	ADD	1	2	1	3			!
1877	UPDATE	1	0	2	1			!
1903	UPDATE	1	0	2	1			!
1164	UPDATE	8	1			R		
48	ADD	1	1	1	1			init
81	ADD	2	2	2	2			!
85	ADD	1	4	1	4			!
109	ADD	3	0	3	0			!
117	ADD	1	1	1	1			init

138	ADD	2	2	2	2				!
142	ADD	1	4	1	4				!
121	UPDATE	3	0	3	0				!
451	UPDATE	1	0	1	0				init
404	UPDATE	2	1	0	3			I	
170	UPDATE	1	0	1	0				init
184	UPDATE	1	0	1	0				!
222	UPDATE	1	0	1	2				init
189	ADD	1	0	1	0				init
201	UPDATE	1	0	1	0				init
223	ADD	1	0	1	0				init
334	UPDATE	1	2	1	2				!
352	UPDATE	1	5	1	5				!
372	UPDATE	1	2	1	0				!
170	ADD	1	0			R			
222	ADD	1	0			R			
31	ADD	2	0	2	0				init
33	ADD	2	0	2	0				init
35	ADD	1	1	2	0				init
37	ADD	1	1	2	0				init
70	ADD	1	2			R			
100	ADD	2	0	2	0				init
30	UPDATE	2	0	2	0				init
32	UPDATE	2	0	2	0				init
34	UPDATE	1	1	2	0				init
36	UPDATE	1	1	2	0				init
69	UPDATE	1	2			R			
100	UPDATE	2	0	2	0				init
375	UPDATE	1	0	3	0				!
507	UPDATE	3	1	4	2				init
518	UPDATE	3	1	4	2				init
564	UPDATE	5	1	4	2				init
570	UPDATE	5	1	4	2				init
577	UPDATE	5	1	4	2				init
205	UPDATE	1	0	0	2			I	
321	UPDATE	1	0	0	1			I	
204	UPDATE	1	1			R			
324	UPDATE	1	1			R			
361	UPDATE	1	0	3	0				!
372	ADD	1	2	1	2				init
203	UPDATE	1	0	1	0				init
491	UPDATE	1	2			R			
561	UPDATE	5	1	6	2				!
570	UPDATE	1	2	0	4			I	
577	UPDATE	5	1	6	2				init
588	UPDATE	5	1	6	2				init
551	ADD	3	1	6	2				init
555	ADD	3	1	6	2				init
630	ADD	5	1	6	2				init
641	ADD	5	1	6	2				init

Table A.3: Commons IO Line Manual Analysis

Appendix B

Missing Test Case Contributions

After we run OPi+ on the systems' commit history, we also manually analyze all files changed more than 100 times, as described in Chapter 4 and Appendix A. For all lines that keep the same number of mutants, we further analyze them and divide them in 2 groups: lines with variables that had no tests for the initialization value and lines for which test cases are missing. For each missing test case, we contributed to the projects with pull requests that add them.

Not all these missing test cases are relevant for the system's functionality. However, an accurate evaluation of the value of these missing test cases can only be made by a system expert. Also, the impact of these contributions are relevant for mutation testing research, not necessarily part of the scope of the continuous mutation testing study. We further describe the contributions made for each project. It is important to note that all folowwing cases have full branch coverage.

B.1 JSoup

Method Not Directly Tested: example for system expert as oracle

The method consumeToIgnoreCase() has full branch coverage. Nevertheless when generating a mutants than cancels out the if(skip==0) all tests pass because the test case is the unique scenario where the behavior on both branches is the same. The if branch states pos++, and the else branch states pos+=skip. In the testing scenario skip is actually 1. By cancelling the if check the result will still be correct but it will require more loops. A change in behavior can be notices only if we run the method twice. Then the same call on the same TokenQueue will return a different string. This behavior is not checked in any other test case, since they all use newly created TokenQueue. Nevertheless we cannot know if this is the intended behavior or not. We provide a test case with the observation that the intended behavior is not explicitly preserved by any test case.

Commit ID 4a470a028e1f146c04695819c48b9ec8b7950f36

Line content: if (skip == 0) // this char is the skip char, but not match, so force advance of pos

Line number: 210

Change type: UPDATE

Survived - Killed Mutants: 1 - 0

Mutator: REMOVE CONDITIONALS MUTATOR

Mutator description: removed conditional - replaced equality check with true

Contribution: Pull request with missing test case[1]

[1] https://github.com/jhy/jsoup/pull/925

Code deleted good refactoring example

The Tag class was first created in this commit. In the beginning the Tag class had 5 boolean attributes. All initialized through a constructor. In the current version of the system, the class evolved to have 8 boolean attributes, all initialized within their declaration. Each true or false value is being explained by a comment. Nevertheless there is no test covering the initialization of this data set.

Commit ID 548ce13435a00bb447fbffdbca8c5ce3be752ee4

Line content:
```
canContainBlock = false;
canContainInline = true;
canContainBlock = false;
canContainInline = false;
canContainBlock = false;
canContainInline = false;
empty = true;
optionalClosing = true;
```

Line number: 239-246, 251-253, 258

Change type: ADD

Survived - Killed Mutants: 2 - 0

Mutator: -

Mutator description: -

Contribution: This is a good example of surviving mutants pointing to a good candidate for refactoring.

Missing test case for wrong intermediate state hasClass()

The method hasClass() from the Element class test if an element has a class. Even though the method is case insensitive it should be able to parse class names given with whitespace. The functionality is there however the if branch that parses this is never tested. Nevertheless for different reasons test cases were added and the mutant status in the last version of the system is 1 - 11.

Commit ID 6e295d4428e4f04baf65c704f00f3142b46f34ff

Line content:
```
if(i-start == classNameLength && classAttr.regionMatches(true, start, className, 0, classNameLength)) {
```

Line number: 1092

Change type: ADD

Survived - Killed Mutants: 3 - 6

Mutator: INLINE CONSTANT MUTATOR; REMOVE CONDITIONALS MUTATOR

Mutator description: Substituted 1 with 0; removed conditional - replaced equality check with true

Contribution: Pull request with missing test case[2]

Duplicated logic by method called reparentChild(...)

The reparentChild() method calls setParentNode() after performing a check. Nevertheless the called method perform the same check. The duplicated logic generates surviving mutants in code that has branch coverage.

Commit ID 674dab0387c4bfad01465574c4be6ea4b3f4f6e9

Line content:
```
if (child.parentNode != null)
```

[2]https://github.com/jhy/jsoup/pull/924

Line number: 277

Change type: ADD

Survived - Killed Mutants: 1 - 2

Mutator: REMOVE CONDITIONALS MUTATOR

Mutator description: removed conditional - replaced equality check with false

Contribution: Pull request with deleted duplicated code[3]

Untested parsing feature convert(...)
The convert() method from W3CDom class, converts a jsoup file into a W3C document. This conversion includes the parsing of the location, into a UIR document, which is never checked. The surviving mutant generated a version of the system that would never add the location. More than this, test were eliminate and the mutant status decreased to 5 - 0.

Commit ID a4883a448416031773bba432bc5bce4492f1e19d

Line content: if (!StringUtil.isBlank(in.location()))

Line number: 55

Change type: ADD

Survived - Killed Mutants: 3 - 0

Mutator: REMOVE CONDITIONALS MUTATOR

Mutator description: removed conditional - replaced equality check with false and true

Contribution: Pull request with missing test case[4]

B.2 Commons Compress

Untested method for octal conversion formatLongOctalOrBinaryBytes(...)
The method formatLongOctalOrBinaryBytes(long value, byte[] buf, int offset, int length) writes into buf the long value as an octal string or as binary. Depending on the length value, it will set a maximum limit against which it checks value. If value is smaller than the limit, it will write it as an octal string, otherwise as a binary number. The surviving mutant removed the check, so buff would always contain the octal string. The existing tests check this method by getting the returned buffer, converting the value within it, back into a long and check if the initial value matches this one. Therefore it would matter if the buffer contained an octal or binary value. This is why the mutant survived with full branch coverage. So, in order to ensure that there is at least one test that will enter the octal string conversion, we added a test that for value = Long.MAX_VALUE with an insufficient small length of 8 for the buffer. This will check that the value can not be converted into an octal and throw an exception with an appropriate message which we check. Nevertheless for different reasons test cases were added and the mutant status in the last version of the system is 3 - 2.

Commit ID: b90b1f445d5b2317360b797afae22ecfccbdac94

Line content: if (length ¡ 9) {

Line number: 370

Change type: UPDATE

Survived - Killed Mutants: 5 - 0

[3]https://github.com/jhy/jsoup/pull/923
[4]https://github.com/jhy/jsoup/pull/922

Mutator: CONDITIONALS BOUNDARY MUTATOR

Mutator description: changed conditional boundary

Contribution: Pull request with missing test case[5]

Outcome: Coverage increased with 0.008% we received a ”Thank you for your patch. I will take a look today.” by one of the top 3 contributors Gary Gregory, in 16 hours.

B.3 Commons IO

Untested method for negative offset value copyLarge(InputStream input, InputStream output, int offset, buffer) The method copyLarge(InputStream input, InputStream output, int offset, buffer) copies the content of input into output. If offset is a positive number, it will skip a number of bytes from input equal to offset value. The surviving mutant came from the offset check, so after looking at the existing tests, we noticed that the copyLarge method was never tested for a negative value for offset. The added test call copyLarge with a negative value and check that the behaviour is the same as for offset = 0, where no bytes are skipped from input.

Commit ID: 57ca8f6d878e3ff0955904cc9e261f254c0320b2

Line content: if (inputOffset ¿ 0) {

Line number: 1483

Change type: UPDATE

Survived - Killed Mutants: 3 - 2

Mutator: CONDITIONALS BOUNDARY MUTATOR

Mutator description: changed conditional boundary

Contribution: Pull request with missing test case[6]

Outcome: Coverage increased with 0.2% we received a ”Thanks!(y)” by the main current contributor in 17 hours.

Commit ID: 57ca8f6d878e3ff0955904cc9e261f254c0320b2

Line content: if (inputOffset ¿ 0) {

Line number: 1633

Change type: UPDATE

Survived - Killed Mutants: 3 - 2

Mutator: REMOVE CONDITIONALS MUTATOR

Mutator description: changed conditional boundary

Contribution: Pull request with missing test case (same as previous)

[5]https://github.com/apache/commons-compress/pull/50

[6]https://github.com/apache/commons-io/pull/41

Appendix C

Selected Object-Oriented Mutation Operators from Literature

This Appendix describes the Object Oriented Mutation Operators proposed in literature and mentioned in this study. The description and examples are based on the original papers where the operators were proposed [48] [42] [43] [61].

C.1 Encapsulation

AMC = Access modifier change
Changes the access level for instance variables and methods to other access levels. [48]

Original Code	**Mutant**
public Stack s;	private Stack s; protected Stack s; Stack s;

C.2 Polymorphism

PMD = Member variable declaration with parent class type
Changes the declared type of an object reference to the parent of the original declared type. [48]

Original Code	**Mutant**
Child b; b = new Child();	Parent b; b = new Child();

PNC = new method call with child class type
Changes the instantiated type of an object reference. [48]

Original Code	**Mutant**
Parent a; a = new Parent();	Parent a; a = new Child();

C.3 Inheritance

IHD = Hiding variable deletion
Deletes a hiding variable, a variable in a subclass that has the same name and type as a variable in the parent class. [48]

Original Code	**Mutant**
class List {	class List {
int size;	int size;
... ...	... ...
}	}
class Stack extends List {	class Stack extends List {
int size;	// int size;
... ...	... ...
}	}

IHI = Hiding variable insertion
Inserts a hiding variable into a subclass. [48]

Original Code	**Mutant**
class List {	class List {
int size;	int size;
... ...	... ...
}	}
class Stack extends List {	class Stack extends List {
	// int size;
... ...	... ...
}	}

C.4 Java-Specific Features Some

JTI = this keyword insertion
Inserts the keyword this. [48]

Original Code	**Mutant**
class Stack {	class Stack {
int size;	int size;
... ...	... ...
void setSize (int size) {	void setSize (int size) {
this.size=size;	this.size=this.size;
}	}
}	}

JTD = this keyword deletion
Deletes uses of the keyword this. [48]

Original Code	**Mutant**
class Stack {	class Stack {
int size;	int size;
... ...	... ...
void setSize (int size) {	void setSize (int size) {
this.size=size;	size=size;
}	}
}	}

JSI = static modifier insertion
Adds the static modifier to change instance variables to class variables. [48]

Original Code	**Mutant**
public int s = 100;	public static int s = 100;

JSD = static modifier deletion
Removes the static modifier to change class variables to instance variables. [48]

Original Code	**Mutant**
public static int s = 100;	public int s = 100;

FAR = Field Access expression Replacement
Replace a field access expression with other field expressions of the same field name [42]. Note that this operator is very broad and was never implemented in a mutation tool or referred in following publications. The following example is based on the operator‘s description, since the authors did not provide any example.

Original Code	**Mutant**

Class C {
private x;
public m {
return x;
}
}
	Class C {
	private x;
	public m {
	return x++;
	}
	}

Original Code	**Mutant**

Class C {
public x;
public m(); {
}
...
A ob1 = new A();
A obj2 = new A();
obj1.x
	Class C {
	public x;
	public m();{
	}
	...
	A ob1 = new A();
	A obj2 = new A();
	obj2.x

MIR = Method Invocation expression Replacement
Replace a method invocation expression with other expressions of the same method name [42]. The following example is based on the operator‘s description, since the authors did not provide any example.

Original Code	**Mutant**
obj.m(1,”abc”)	obj.m(2,””)