OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Taneli Taipale

IMPROVING SOFTWARE QUALITY WITH SOFTWARE ERROR PREDICTION

Master's Thesis
Degree Programme in Computer Science and Engineering
December 2015

Taipale T. (2015) Improving software quality with software error prediction. University of Oulu, Department of Computer Science and Engineering. Master's thesis, 71 p.

ABSTRACT

Today's agile software development can be a complicated process, especially when dealing with a large-scale project with demands for tight communication. The tools used in software development, while aiding the process itself, can also offer meaningful statistics. With the aid of machine learning, these statistics can be used for predicting the behavior patterns of the development process.

The starting point of this thesis is a software project developed to be a part of a large telecommunications network. On the one hand, this type of project demands expensive testing equipment, which, in turn, translates to costly testing time. On the other hand, unit testing and code reviewing are practices that improve the quality of software, but require large amounts of time from software experts. Because errors are the unavoidable evil of the software process, the efficiency of the above-mentioned quality assurance tools is very important for a successful software project.

The target of this thesis is to improve the efficiency of testing and other quality tools by using a machine learner. The machine learner is taught to predict errors using historical information about software errors made earlier in the project. The error predictions are used for prioritizing the test cases that are most probably going to find an error.

The result of the thesis is a predictor that is capable of estimating which of the file changes are most likely to cause an error. The prediction information is used for creating reports such as a ranking of the most probably error-causing commits. Furthermore, a line-wise map of probability of an error for the whole project is created. Lastly, the information is used for creating a graph that combines organizational information with error data. The original goal of prioritizing test cases based on the error predictions was not achieved because of limited coverage data. This thesis brought important improvements in project practices into focus, and gave new perspectives into the software development process.

Keywords: error prediction, continuous integration

Taipale T. (2015) Ohjelmiston laadun parantaminen ohjelmointivirheiden ennustamisella. Oulun yliopisto, Tietotekniikan osasto. Diplomityö, 71 s.

TIIVISTELMÄ

Nykyaikainen ketterä ohjelmistokehitys on monimutkainen prosessi. Tämä väittämä pätee varsinkin isoihin projekteihin. Ohjelmistokehityksessä käytettävät työkalut helpottavat jo itsessään kehitystyötä, mutta ne myös säilövät tärkeää tilastotietoa. Tätä tilastotietoa voidaan käyttää koneoppimisjärjestelmän opettamiseen. Tällä tavoin koneoppimisjärjestelmä oppii tunnistamaan ohjelmistokehitystyölle ominaisia käyttäytymismalleja.

Tämän opinnäytetyön lähtökohta on ohjelmistoprojekti, jonka on määrä toimia osana laajaa telekommunikaatioverkkoa. Tällainen ohjelmistoprojekti vaatii kalliin testauslaitteiston, mikä johtaa suoraan kalliiseen testausaikaan. Toisaalta yksikkötestaus ja koodikatselmointi ovat työmenetelmiä, jotka parantavat ohjelmiston laatua, mutta vaativat paljon ohjelmistoammattilaisten resursseja. Koska ohjelmointivirheet ovat ohjelmistoprojektin edetessä väistämättömiä, on näiden työkalujen tehokkuus tunnistaa ohjelmointivirheitä erityisen tärkeää onnistuneen projektin kannalta.

Tässä opinnäytetyössä testaamisen ja muiden laadunvarmennustyökalujen tehokkuutta pyritään parantamaan käyttämällä hyväksi koneoppimisjärjestelmää. Koneoppimisjärjestelmä opetetaan tunnistamaan ohjelmointivirheet käyttäen historiatietoa projektissa aiemmin tehdyistä ohjelmointivirheistä. Koneoppimisjärjestelmän ennusteilla kohdennetaan testausta painottamalla virheen todennäköisimmin löytäviä testitapauksia.

Työn lopputuloksena on koneoppimisjärjestelmä, joka pystyy ennustamaan ohjelmointivirheen todennäköisimmin sisältäviä tiedostomuutoksia. Tämän tiedon pohjalta on luotu raportteja kuten listaus todennäköisimmin virheen sisältävistä tiedostomuutoksista, koko ohjelmistoprojektin kattava kartta virheen rivikohtaisista todennäköisyyksistä sekä graafi, joka yhdistää ohjelmointivirhetiedot organisaatiotietoon. Alkuperäisenä tavoitteena ollutta testaamisen painottamista ei kuitenkaan saatu aikaiseksi vajaan testikattavuustiedon takia. Tämä opinnäytetyö toi esiin tärkeitä parannuskohteita projektin työtavoissa ja uusia näkökulmia ohjelmistokehitysprosessiin.

Avainsanat: ohjelmistovirheiden estimointi, jatkuva integraatio

TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

ABBREVIATIONS

1. INTRODUCTION 7

2. BACKGROUND 9

2.1. Software development 9
2.1.1. Development models 10
2.1.2. Development practices 12
2.2. Testing 16
2.2.1. Test coverage 16
2.2.2. Testing levels 18
2.2.3. Regression testing & test automation 21
2.3. Software error prediction 21
2.3.1. Feature data 23
2.3.2. Classifier data 26
2.3.3. Machine learning 28
2.3.4. Predictive testing 30

3. PROBLEM : RESOURCE EFFICIENCY IN TESTING 32

4. DESCRIPTION OF SOLUTION 35

4.1. Feature extraction: raw data & pre-processing 37
4.1.1. Version control history 39
4.1.2. Static software metrics 40
4.1.3. Peer review and build history 40
4.1.4. Organizational metrics 41
4.2. Error data 41
4.2.1. Raw error data: Issue tracking 41
4.2.2. Error data pre-processing: Estimation of fix-inducing code . . 42
4.3. Statistical model 45
4.3.1. Model selection 46
4.3.2. Feature selection 46
4.4. Reporting 48
4.4.1. File change hotness ranking 48
4.4.2. Code hotness graph 49
4.4.3. Error-developer graph 50
4.5. Test case prioritization 52

5. ACHIEVEMENTS **53**
5.1. Error graph 54
5.2. Publications 54

6. FUTURE WORK **56**
6.1. Improvements in feature data 56
6.1.1. Unit testing as a feature 56
6.1.2. Filtering of code changes 56
6.2. Test case prioritization 57
6.3. Error prediction in continuous integration 57

7. DISCUSSION **59**
7.1. Challenges in data 59
7.1.1. Obvious output data 59
7.2. Threats to validity 60
7.3. Good and bad practices in software development 60
7.4. Lessons learned 61
7.4.1. Sensitive information 62

8. CONCLUSIONS **63**

9. REFERENCES **64**

10. APPENDICES **70**
10.1. SZZ algorithm 70

FOREWORD

This thesis represents the work made in Bittium (former Elektrobit). The main instructor for this thesis was Timo Räty. The periodic light bulb moments would probably have not surfaced without the invaluable feedback and wise words he gave. This thesis was made under the guidance of Prof. Burak Turhan. I would like to thank Burak for his ideas and support for making the error prediction model and for the initial spark he gave for this project. Mika Qvist helped shaping the final predictor. He also helped to shape my understanding of software quality, a very vital trait considering the topic of this thesis. Thank you goes also to my supervisor Prof. Juha Röning and the second inspector Lauri Tuovinen. Finally I would like to thank Jesse Pasuri for the enthusiasm in the topic and for the occasional kick in the butt to get the thing over with.

Oulu, Finland November 30, 2015

Taneli Taipale

ABBREVIATIONS

API	Application Programmable Interface
ARM	Advanced RISC Machines
ASIC	Application Specific Integrated Circuit
CC	Cyclomatic complexity
CI	Continuous integration
CPU	Central processing unit
CVS	Concurrent Versions System
FPGA	Field-programmable gate array
LOC	Lines of code
REST	Representational State Transfer
RISC	Reduced Instruction Set Computer
SCM	Source Code Management
SSH	Secure Shell
SW	Software

1. INTRODUCTION

Ever-increasing complexity has forced the software process to change [1, 2]. On the one hand, the transition from waterfall development methods to agile methods has made software development more adaptable to suddenly changing conditions. On the other hand, sudden changes and the complexity of software have also caused software development to become a more difficult process to handle. Ways to see into a software process would be valuable. The data needed for these viewpoints are now available in the tools that have been created to support the development process [3, 4].

Testing is an inseparable part of software design. As software development methods have advanced, testing has also had to evolve [5, 6]. Test automation is possibly the only way to cope with the need of constant testing of the working increments made by an agile software team. Continuous integration is the culmination of agile software development and testing, as it aims to integrate, build and test the whole system for every commit. Continuous deployment takes continuity one step further by also automating the process of deploying the software to the customer.

Regression testing tries to guarantee that any change made in the software does not cause any failures in the system. Testing the whole system is costly. Testing the system as whole often causes the tests to run for a long period of time. Furthermore, the related testing equipment is also expensive. Test case prioritization is a widely researched way to improve testing efficiency. The idea in test case prioritization is to somehow weigh the value of test cases and choose an order that produces an optimal testing outcome, i.e. the maximal amount of errors is found in the minimal amount of time. Any information as to which of the tests are needed would be valuable.

The large codebase is a problem as the code will contain errors and require a lot of testing and code reviewing. On the one hand, code reviewing is an effective practice for finding errors in the early stages of development. On the other hand, even when using modern code review tools, it costs expert coders time that they could use to be effective in other ways. A proper amount of code reviewing balances the price of otherwise missed errors with the effort put into reviewing the code. Pinpointing the problem areas would be a way to make code reviews more effective.

Error prediction is a potential method to tackle some of the problems of ever-growing software projects [7, 8, 9, 10]. It aims to predict where the errors will occur. This information can then be used to target testing and code reviewing. Error prediction gives viewpoints on the process of creating software that otherwise could not be seen.

The goal of this thesis is to create a software error predictor for an embedded software project at Bittium (formerly known as Elektrobit). The error predictor should be able to predict errors correctly, so that the error prediction information can be used for increasing the quality of the software. The prediction information shall be used for creating reports that give practical benefit to developers. These reports help answer questions such as "Which code changes should be reviewed most thoroughly?" and "Where should future unit testing efforts be concentrated?"

Another goal of this thesis is to improve testing efficiency by using information as to where the errors most likely exist. The tests should be run based on the information as to how probable it is that the test will find an error. By concentrating the testing efforts to tests that are most likely to find the errors, expensive physical resources and time can be saved. This can be done while maintaining or even improving the testing quality.

Modern software project management and development tools that are used in most enterprise software projects provide transparency and a constant data source to be mined. These data can be used for effective error prediction.

2. BACKGROUND

Software error prediction measures the software development process in order to create predictions. The change in software development processes towards agile is described in this chapter. The change in the software development process is the motivation for the current setting of development tools and methods. These software development tools are examined in order to gain measurements for the error prediction model.

A background of software testing is given. Software testing is one of the most important practices regarding software quality. One of the goals of the error predictor is to make testing more efficient by using test prioritization. Test automation and continuous integration are practices that make it possible to have a continuous flow of development and an error prediction mechanism to realize.

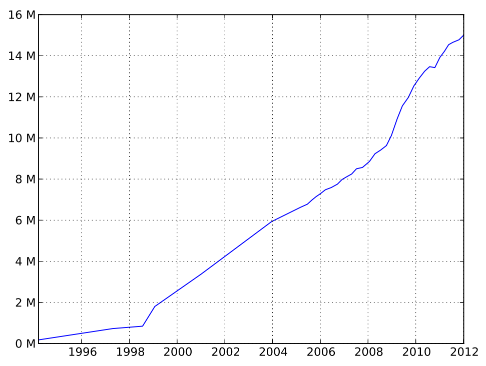
Finally, the background for software error prediction is given. The different features that are measured about software development are described. Existing machine learning algorithms, which form the mathematical base for error prediction, are presented.

2.1. Software development

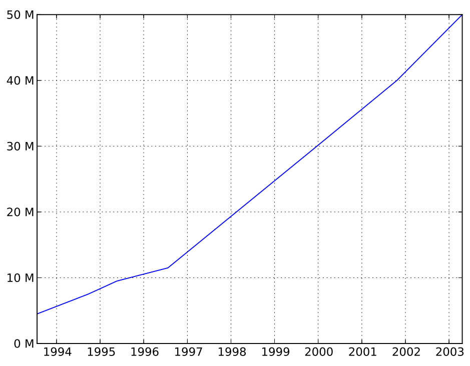
Software development has been around from the middle of the 20th century [11]. Since then, the software business has been optimizing software practices in an attempt to maximize the quality of the software produced. One major concern in software development has been the development model. Different development models have been evaluated to maximize the value of the software that has been developed.

Some advancements have also been made in development practices. Customer requirements are now stored and conformed to in a traceable and transparent manner. This is done by using requirement management and issue tracking. Development practices such as continuous integration and continuous deployment have emerged and been taken into use just recently. The already established practices, such as code reviewing, have gained new value from the automated tools created to support them. Guido van Rossum has noted from hands-on experience: "proper code review habits can really improve the quality of a code base, and good tools for code review will improve developers' life" [3].

The need for more advanced development has risen from the ever-growing size and complexity of produced software projects. In Figure 1 the lines of code in the Linux kernel [12, 13, 14] and Windows [15] are plotted over time. Windows and the Linux kernel are both large operating system software projects. Windows is an example of an enterprise project and Linux an example of an open source project. Both of these projects are suitable for describing software development in general. As can be seen, over time the number of lines of code is constantly growing for both the enterprise and the open source project. This growth in the number of lines of code also causes a growth of complexity. The constant growth of complexity causes these projects to get more challenging to maintain. This kind of growth in the amount of code has caused a demand for new and more advanced development methods.



(a) Lines of code in the Linux kernel over time



(b) Lines of code in the Windows operating system over time

Figure 1: The growth of software projects

2.1.1. Development models

Earlier software development models were based on the waterfall model [16], seen in Figure 2. The waterfall model relies on development progress that goes sequentially through phases of requirement specification, analysis, design, coding, testing and operations. Even in the earliest documents that describe the model, waterfall is deemed to be risky. The testing phase is the first phase where the effects of the real world can be experienced [16]. At that point, the errors made in earlier phases manifest. For example, the errors can be a result of unclear requirements. The errors made in earlier phases have to either be fixed by design choices or by inducing iteration, the latter requiring less effort. Iteration is something the waterfall model has not taken into account. While later development models replace the working order of the waterfall model, the original development phases stay relatively the same.

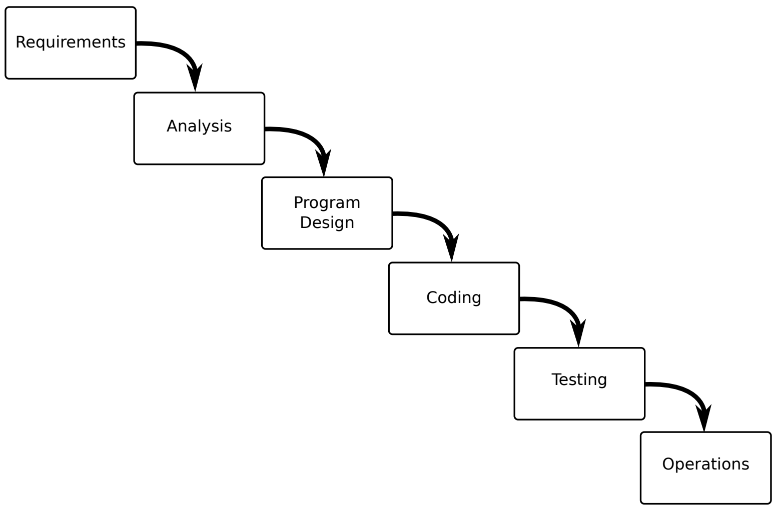


Figure 2: The waterfall model

Iterative models have been used as a replacement to the waterfall model as early as the mid-1950s [1]. They are more realistic in the sense that they take into account the possible errors made in different phases of design and implementation. Iterative models try also to alleviate the problems that arise from unclear or changing requirements. Iterative development works by going iteratively through phases of planning, design, testing and evaluation, while producing iterations of the final product that constantly improve upon the earlier iterations. An example of an iterative model is described in Figure 3[17].

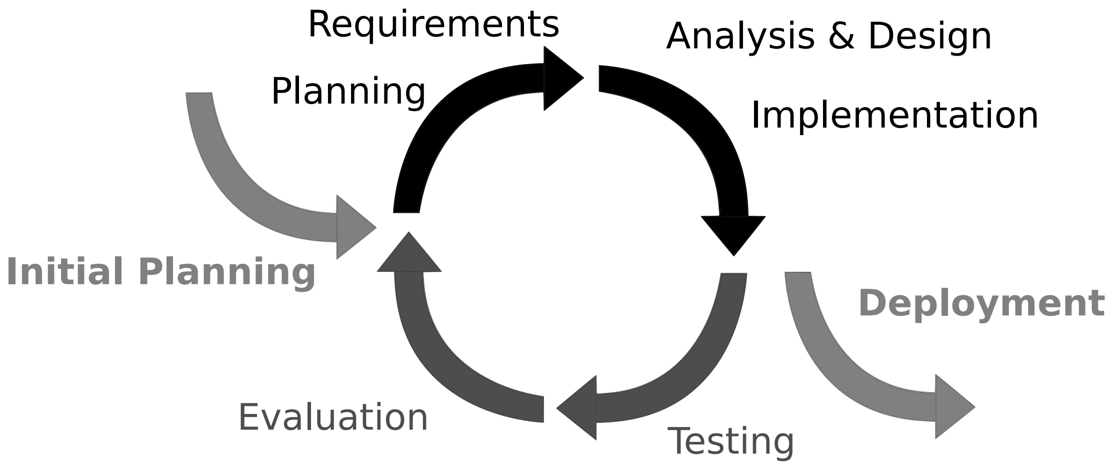


Figure 3: An iterative model

Agile is a development methodology that has gathered a lot of popularity [2] since its creation in the year 2001. Agile is based upon the manifesto made by Kent Beck et al. [18]. It promotes creating quality software with less emphasis on strict processes and planning. "Agile methods" is a term that describes a collection of methodologies that extend upon the iterative model [1], such as Scrum, Extreme Programming and Feature Driven Development.

Scrum [2] is the most popular of the agile methods. It describes an iterative project development methodology that is carried out by a single cross-functional team [19]. Efforts in Scrum are carried out in a timeboxed manner, described in Figure 4. Modern development practices such as continuous integration, requirement management and issue tracking often build upon or are related to Scrum.

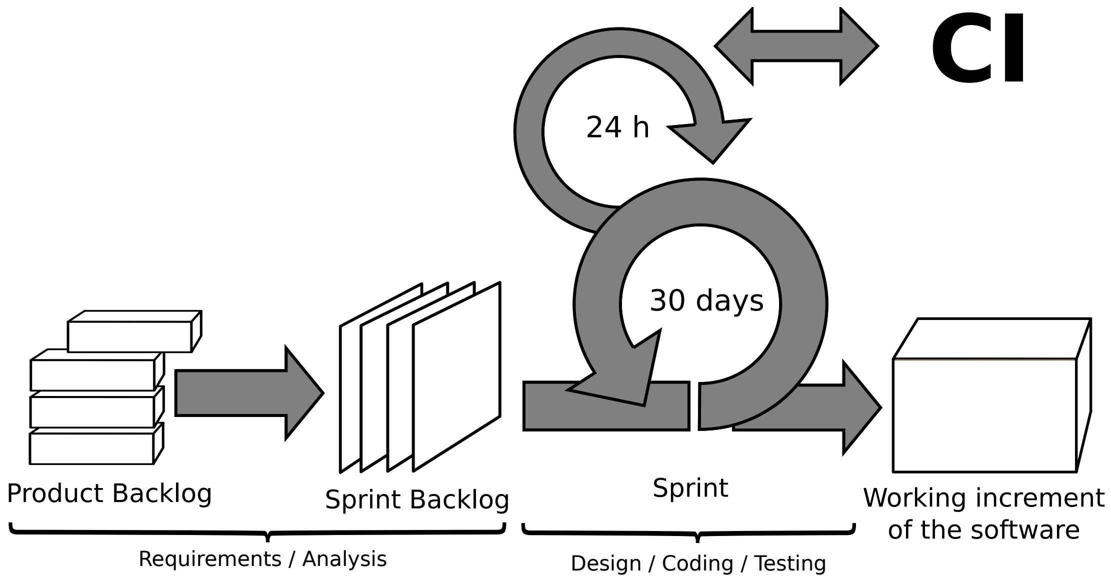


Figure 4: The Scrum process

2.1.2. Development practices

Software projects have a certain palette of development practices. The most basic of them are listed in Table 1.

Table 1: Development practices and the tools that support them

Development practice	Supporting tools
Continuous integration and deployment	ThoughtWorks Go, Jenkins, CruiseControl
Issue tracking & Requirement management	Bugzilla, Atlassian Jira, Trac
Version control	Git, CVS, SVN
Peer review	Gerrit, Crucible, Github

The complexity of the software project dictates the extent to which these practices are utilized. For example, a small project might cope without a well-defined issue tracking and with no continuous integration system in place. However, if a project is big and complex, these practices are vital.

The above-mentioned development practices have tools to support them. These tools have been made to make it easier to take these development practices into use. They also have the ability to store the whole activity history. A combination of the data stored by these tools can show quite a comprehensive history of a project.

Continuous integration and deployment

The size of a change goes hand in hand with the time required to integrate it into a product. Merging is the act of integrating code changes that have been done in different workspaces. The purpose of merging is to produce or maintain the desired functionality of the software. When dealing with bigger merge efforts, the time spent integrating increases exponentially. This can cause "integration hell," an anti-pattern often cited in Extreme Programming, the parent principle of continuous integration. Integrating smaller packages frequently leads to a system that is able to quickly detect integration problems. This is the main reason continuous integration exists.

Continuous integration is described by Martin Fowler as "...simple practice of everyone on the team integrating frequently, usually daily, against a controlled source code repository." [20] A basic continuous integration scenario is described in Figure 5, made on the basis of the work of Duvall [21 p. 5]. Continuous integration automates repetitive tasks such as testing and building. By doing that, it speeds up the software development process.

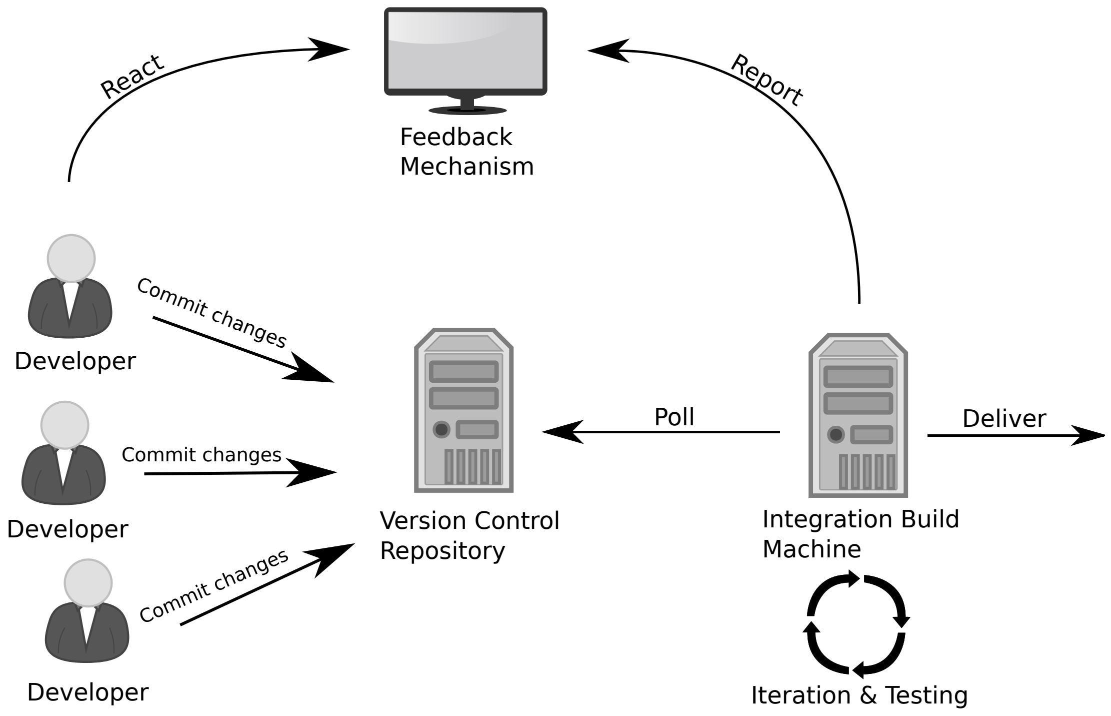


Figure 5: A basic continuous integration scenario

In a basic continuous integration scenario, developers push commits into a mainline through a version control repository. At that point, developers are responsible for merging their efforts with other developers' efforts. An integration build machine then builds a working copy from the latest head of the mainline. If the build succeeds, the produced build is directed to the upstream. Otherwise, if the build does not succeed, developers are required to fix the product before integration is resumed. The feedback given to the developers is an important part of continuous integration. The purpose of failing fast and detecting integration problems early is to get this information back to the developers as quickly as possible.

Modern continuous integration systems often include testing that also gives instant feedback. By integrating more and more testing, continuous integration systems are slowly shifting towards continuous deployment. Continuous deployment is also known as continuous delivery. In continuous deployment, the product is at such maturity in every iteration step that the product can be deployed or released to the customer. The steps of continuous deployment are shown in Figure 6 based on work of Humble and Farley [22].

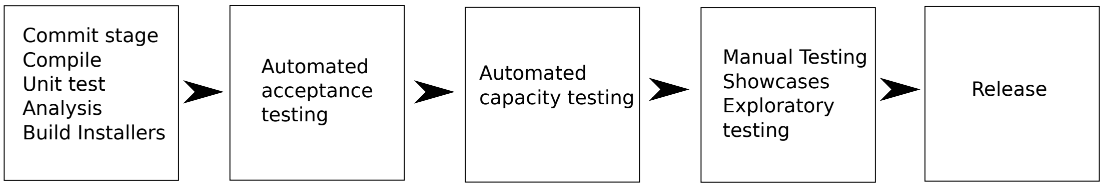


Figure 6: Steps of continuous deployment

Issue tracking & requirement management

Product planning and development activities can be brought down into implementing requirements and issues.

Requirements describe the planned product features. They are set at the planning phase of the product, but can also be altered during development. Requirements have the important function of being the basis for testing. Test cases are created on the basis of required functionality. In that sense, requirement management is of equal importance to the actual development of the software and to the testing actions surrounding it.

Issue tracking is the practice of following the developer efforts and customer issues in a well-defined manner. Issue tracking also accounts for tracking the errors already found in the software. The motivation of issue tracking is to make sure that every issue is acted upon. For example, issues brought up by separate customers can be bundled up. Issue trackers open to the public can also save the customers' time because the status of common issues is visible to all of them. This is the case with many open source projects, such as the ones on GitHub [23].

Version control

Version control is the practice of maintaining a history of a software development project. It enables developers to collaborate on a shared codebase. The key benefits of using version control are reversibility, concurrency, and annotation [4].

Reversibility enables the developers to reverse into a previous edition of the software. This is important for example when something has gone wrong in the development process and the last working version is needed.

Concurrency is one of the main functions of a version control system. Concurrency gives developers the ability to work on the same project simultaneously in multiple locations. Concurrency is made possible by the merging of two separate branches in a controlled way. This is done when two different branches of software have deviated due to work done in isolation. The act of merging is fairly simple when there are no dependencies between the files that have been changed in the different branches, and the changes have been made in separate locations inside the files. Otherwise the merging demands manual work from the developer. The merging operations can produce errors and that is one of the motivations for thorough regression testing.

Annotation works as a way of keeping track and commenting the changes in version control history. The means of annotation for a typical version control system are commit messages and tags that are attached to commits. The minimum annotation that version control enforces is the splitting of work effort into commits.

The evolution of version control software can be divided into three generations shown in Table 2 [4].

Table 2: The three generations of version control

Generation	Mode of operation	Examples
1st	Local	SCCS, ClearCase
2nd	Centralized	CVS, Subversion (SVN)
3rd	Decentralized	Git, Mercurial

The earliest version control scheme was localized version control. In local version control, operations are allowed only to a single file at a time in a local repository. The second generation, centralized version control, allows a client-server type of architecture. The third generation decentralizes version control so that it is possible to have separate versions of the repositories in different locations.

Annual surveys conducted by the Eclipse Foundation [24, 25] show that the third generation version control systems are close to being the most popular. The share of third generation version control systems in the year 2014 was 48.9%. In comparison, the share was only 3.5% in the year 2009. One of the reasons for the rising popularity of the third generation version control systems is scalability. Decentralization means that the load is distributed away from the master. The master server is contacted only in two cases: either the changes are desired to be made public, or the latest changes are fetched. This makes the system scalable for a very large number of remote working spaces.

Peer review

In addition to testing, peer review is one of the principal methods of assuring code quality. It has been also proven to be very effective [26, 27]. Basically, peer reviewing is the practice of having professionals review each other's written source code. Peer reviewing can be separated into formal and informal, or lightweight, code reviewing. Informal code reviewing tends to be more efficient when considering the time invested [27].

The availability of tools for code reviewing has made informal, lightweight code reviewing popular. Current source code collaboration systems such as GitHub and BitBucket [23, 28] have code reviewing tightly merged to code committing workflow. Open source projects have an even tighter connection with peer reviewing and the overall workflow. Linux is an example of how persistent peer review has created an operating system that is known for its long-term stability.

2.2. Testing

Testing is the cornerstone of software quality. Defined testing methods are key quality assurance tools to ensure requirement conformity. Evidence which proves software to be conforming to its requirements is important for both the customer and the responsible software project. A test is a four-phase task [29] that consists of:

- Setup
- Exercise
- Verification
- Teardown

The exercise and verification steps of a test are best described by test coverage. Test coverage is an important concept in testing, and it is an application of graph coverage for the program code.

Testability of software is an aspect that has to be taken into account from the beginning of designing software. It is the extent to which the software supports testing. For example, requirements for the software affect how testable the software ultimately is. Testing can be divided into separate testing levels that test the different layers of software development. Figure 7 [30] describes how design and development activities affect each of the testing levels.

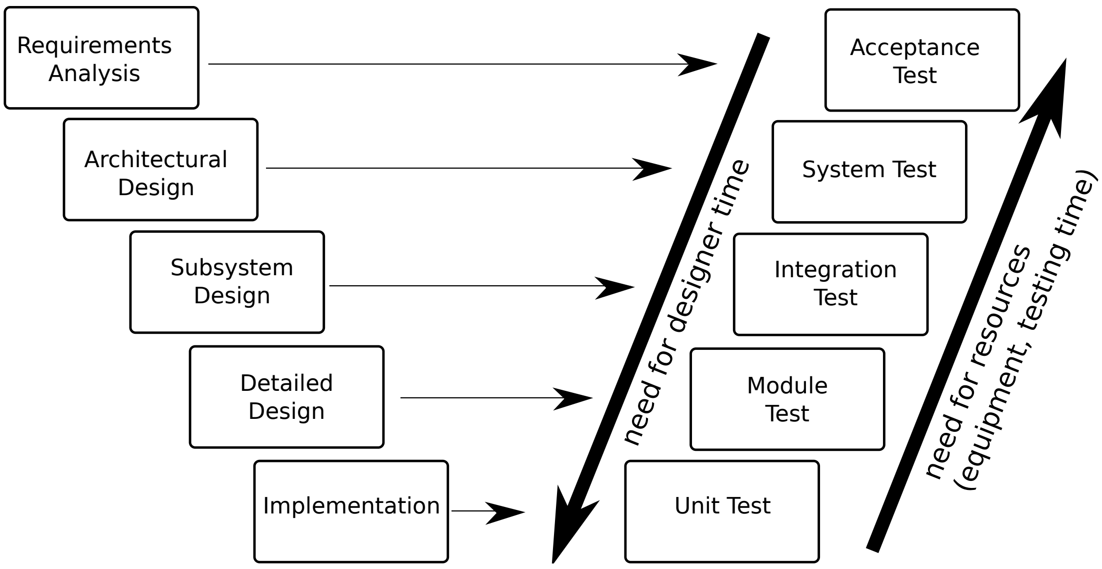


Figure 7: Relations between software design and development activities with testing

2.2.1. Test coverage

A program under test can be expressed as a control flow graph, as described by Ammann and Offut [30]. A statement or a branching point in the program is a node in the graph. The edges in the graph are then the branches of the program. Source code coverage is then the practical application of graph coverage, where each node is a statement or a source code line in the program. The amount of functionality a test covers can then be measured in different types of coverage criteria such as:

Statement coverage The amount of nodes that are covered. For source code, statement coverage is usually simply referred to as code coverage.

Branch coverage The amount of branches that are covered. Full branch coverage means that every outcome of a branching point is visited.

Path coverage The amount of paths that are covered. Achieving full path coverage means that every **combination** of branching point outcomes are visited.

Figure 8 shows how these coverages are achieved by choosing different paths of execution.

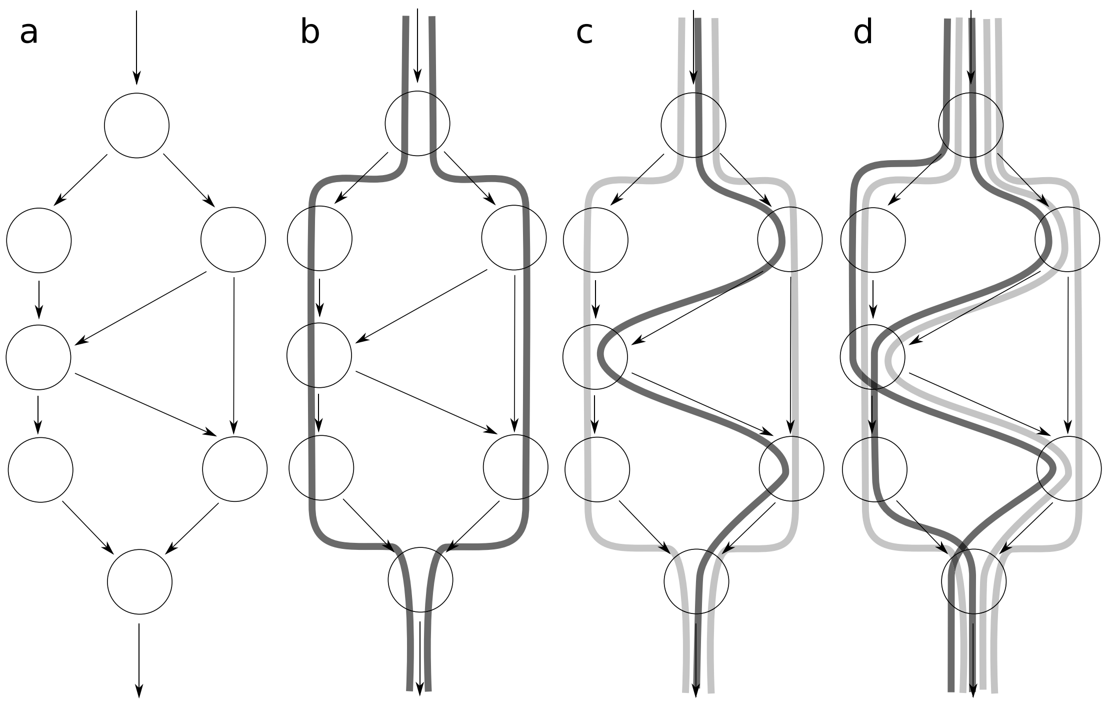


Figure 8: a) an example control flow graph and sets of paths that achieve full b) statement, c) branch and d) path coverage

Source code coverage is easy to determine in unit testing and it describes the coverage in an understandable manner. Source code coverage for unit tests can be gathered with code instrumentation, and tools are available for many of the programming languages [31]. It is feasible to aim for full statement, branch or path coverages in unit testing. In higher level tests, the meaning of code coverage becomes more blurred, and determining when a full code coverage is reached will be hard. The instrumentation needed for code tracing might be impossible, or the instrumentation itself would affect the system under test so that the tests would be impossible to execute.

The correlation between thorough unit testing and quality has led to the common use of code coverage as a quality metric. For example, monitoring the trend of code coverage can tell if the testing keeps up with implementation. Code coverage on its own can still be a misleading metric of quality, as good testing leads to high coverage, but high coverage does not necessarily imply a good quality of testing.

Coverage is created at the exercise phase of the test. For coverage, it does not matter if anything is verified at the verification phase. This means that a 'null' test could show a 100% code coverage but still assert nothing when verification is missing. In other words, such a test does not fulfill the four testing phases.

2.2.2. Testing levels

Testing can be divided into separate layers. The most usual way to partition them, described amongst others by Ammann and Offutt [30], is:

1. Unit testing
2. Module testing
3. Integration testing
4. System testing
5. Acceptance testing

Unit testing is the fastest testing level, has the least expensive equipment needed and is able to scale if needed. When going further up the testing levels, these qualities go towards the contrary. Acceptance testing at the other end is very slow, requires expensive equipment and does not scale well.

All of these testing levels work on the basis that the software under inspection has been tested in the previous, lower testing level. Testing on a level higher than the actual maturity level of the software is asking for trouble, as passing down the levels to eventually find the problem is time-consuming.

Unit testing

Unit tests test the code on the lowest level possible. Figure 9 shows how they test that a certain function or method produces the required output with the given input. They run in a short amount of time. Unit tests can be run in the developer's own environment, without the need for a real hardware or software environment. Fast and effortless to run, unit tests work as an important tool for developers who are working on a collaborative code base.

Unit tests are often time consuming to implement. The amount of unit test code can typically rise well above the amount of production code [32, 33]. The cost of creating unit tests is justified by the fact that unit level is the cheapest level to detect defects on. Test-driven development is the practice of creating tests before creating code that passes them [5]. This method has been proven to be effective in improving the overall quality of software [6].

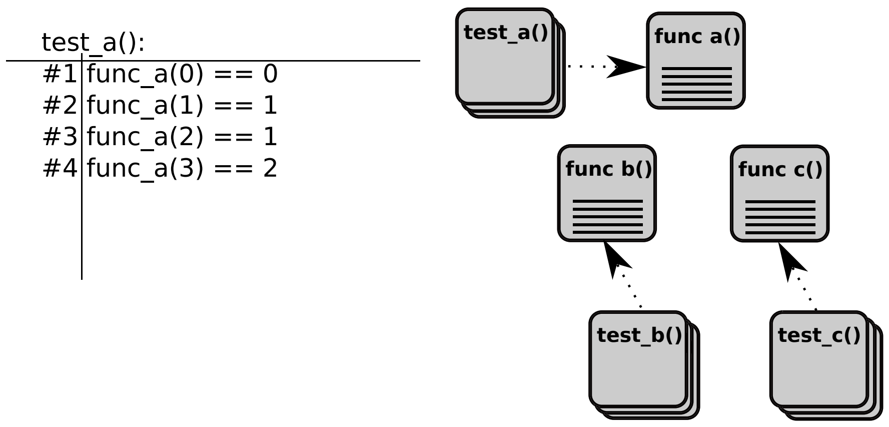


Figure 9: Unit testing

There are also other elements of difficulty in unit testing. Unit tests should not call for a setup of surrounding interfaces, and they should not require a certain program state. That is why unit tests usually need stubbing and mocking of program states and outside interfaces. However, unit tests encourage the developer to create code that is testable. Usually this means that a separation has to be done between the code that has side-effects, such as external input and output, and the functional code. This separation then makes it easier to test only the functional part and reduces the need for mocking.

Module testing

Module tests verify that the module works correctly. Figure 10 gives an idea how module tests work. They test that a collection of functions and/or methods work properly together. The definition of a module is dependent on the programming language that is used. For object-oriented languages, the term "module" means a class and for non-object-oriented languages, it means a file or a package.

Module testing is not concerned with the implementation of individual functions but how they work together. Module tests take more time to execute than unit tests, but should still run in a relatively short time. Module tests can often be run in a virtual environment to save the cost of a real hardware environment.

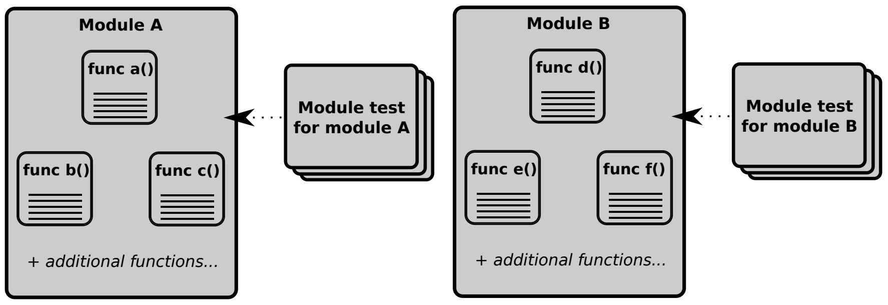


Figure 10: Module testing

Integration & system testing

Figure 11 shows how integration tests run different modules, possibly made by different teams or developers.

Integration tests aim to seek errors between module interfaces. Integration testing also finds errors that are only possible to detect when threading, I/O and such outside factors come into play.

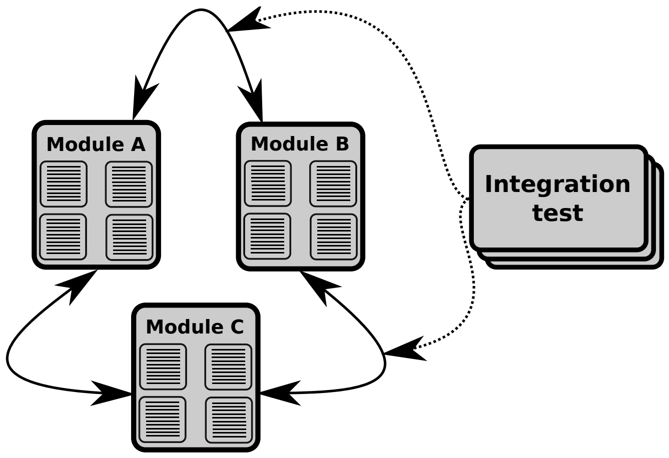


Figure 11: Integration testing

Figure 12 presents system testing. System testing is similar to integration testing as it also tests module interfaces. System testing verifies that all of the modules work together to provide a fully functional system. System testing seeks to find problems arising from mistakes made in planning and specification.

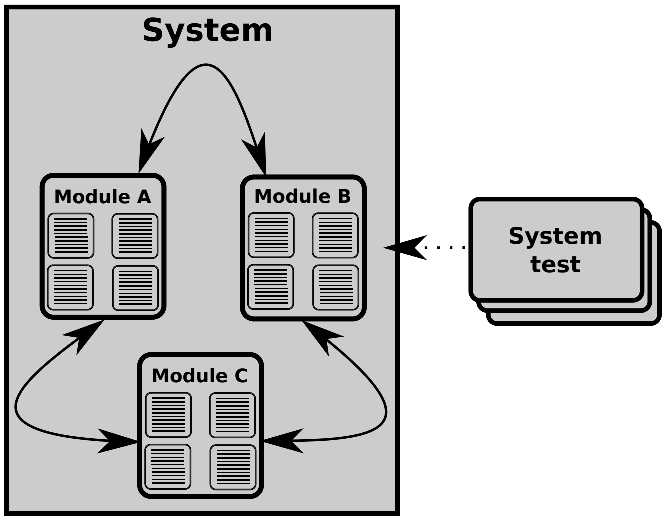


Figure 12: System testing

Both integration and system testing need a test setup that consists of, for example, needed databases and possible hardware. They also run a significant amount of code. That is why they take a long time to run and are expensive.

Acceptance testing

Acceptance testing tests the system in an environment that imitates the real world as closely as possible. This testing level needs more expertise than the other levels concerning the environment the software is deployed to. In comparison to system testing, which tests the system against developer-created acceptance criteria, acceptance testing ensures that the system really does what the user wants. As acceptance testing is usually done by a separate testing team in a separate organization, finding errors is very expensive. The testing time and equipment already cost a lot. In addition to those, the errors are found on a very high level, so the amount of code that can be the cause of the error is vast.

2.2.3. Regression testing & test automation

Continuous integration tries to keep the current version of the software as functional as possible. This is done by gate keeping the version control system, i.e. commits are tested to decide whether they should be integrated into the product or not. Testing every integration step requires a great quantity of tests. If the tests were executed only by human labor, they would require a huge amount of it. The testing goal of continuous integration can be achieved far more effortlessly by using test automation. Test automation is the practice of creating, or translating existing manual tests to, machine-runnable tests.

Of the different types of testing levels, unit tests are the cheapest and fastest to automate. The need for outside software and hardware rises when going up the testing levels. Tests on upper levels are more expensive to automate, as they involve setting up a complex environment, or acting out a complicated use case.

Test automation can be implemented on different levels of abstraction. The most basic test automation can be a recording of events made into a runnable script that is then played back to test the product. A more advanced test system can use a model to vary the test data. Model-based testing is currently the most sophisticated way of automating tests. In addition to running the tests, a model-based tester can also automate the task of designing the tests [34].

2.3. Software error prediction

The goal of software error prediction is to predict how probable it is that a certain part of a project causes a failure. The probability can be estimated for lines of code, a component assembly or any other kind of development package. This information can be used for purposes such as:

- Targeting code reviews
- Targeting testing
- Targeting project communication

In addition to the items listed above, the information about the mathematical or algorithmic model used for prediction itself can be used for studying the habits of the project. It must be noted that this approach requires the model not to be too obscure. As an example an error prediction based on N-variable linear regression is shown in Equation 1. Here Y is the probability of errors and X are the values of the features. The fitted values of the estimated weights β would provide the most probable cause for errors.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_N X_N + \epsilon_i \tag{1}$$

Implementing error prediction can be seen as a supervised machine learning classification problem. Supervised machine learning is the practice of "training" a machine learner based on historical data. The trained machine learner can then classify the current data at hand. This means fitting an algorithm or a mathematical formula to predict from a number of features the correct class for input data. For classification problems, the historical data need to have correct classifications included.

The basic unit of the software development history is a commit. For the commits, the features are extracted and the classification is made. The process of error prediction is presented in Figure 13.

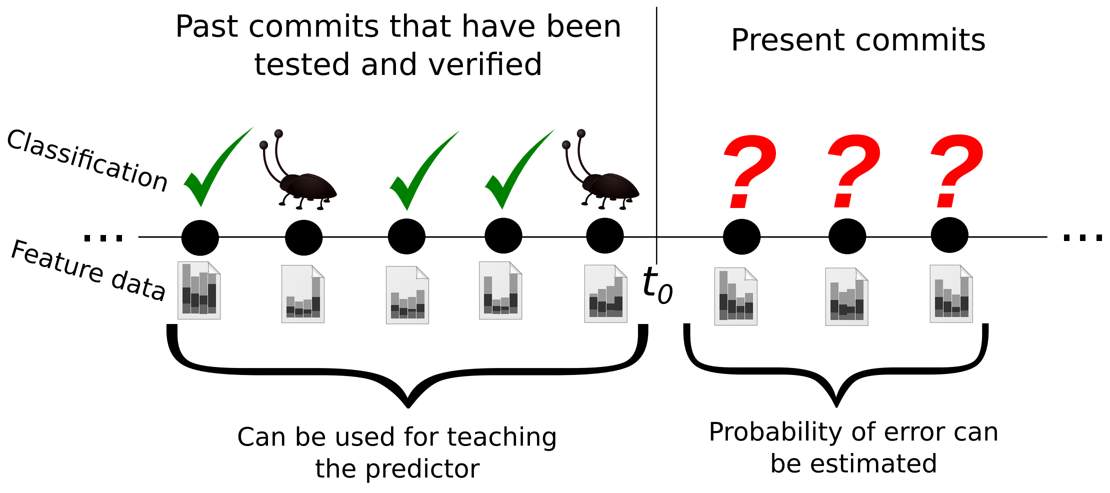

Past commits that have been tested and verified
Present commits
Classification
Feature data
t_0
Can be used for teaching the predictor
Probability of error can be estimated

Figure 13: Predicting commit faultiness using machine learning

In Figure 13, t_0 is a moment in time for which these statements hold true:

- Commits made before time t_0 are used for training the predictor.
- The probability of error is predicted for commits made after the time t_0.

Before the time t_0, it is possible that some of the commits have errors that have not yet been reported. Those commits are still marked as error-free commits by the error predictor. This creates some inaccuracies in the predictions. The decision of where to place the time t_0 is based on the amount of time a commit should exist before it can said to be thoroughly tested. However, choosing the time t_0 to be too far in the past will cause the predictions to be outdated, as they miss the latest knowledge of the erroneous commits.

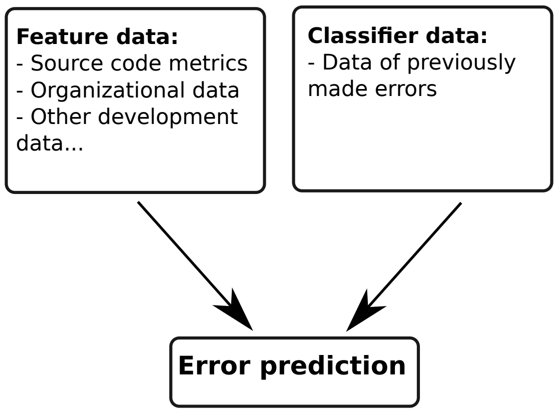


Figure 14: Machine learning

In software error prediction, the training data are typically the history of the project at hand. The use of combined histories of earlier projects has also been studied [35, 9]. Error prediction needs feature and classification data. In a typical software project, features that can be extracted are information about the code, organization and the development history. Classifier data are data about which parts of the project have been faulty. Using both the feature and classifier data, an error predictor can be created. This idea is presented in Figure 14.

2.3.1. Feature data

Feature data consists of individual heuristics that describe the process in some way. Ideally in error prediction, these features would correlate with errors or explain the errors. A great amount of the research concerning error prediction has been done to find optimal features that would best correlate with errors [7, 10, 36, 37, 8, 38, 39].

Most of the software error prediction models made in the past have used static software quality metrics to predict software faultiness [7]. Newer approaches such as using features derived from change history [10] or mapping the authors of these changes to an organizational tree [36] have also been used. One interesting new approach to predict errors is the use of micro interaction metrics, which try to capture the most subtle developer behavior patterns [37].

Static software metrics

Static software metrics express the code in some measurable way. Most of the static software metrics tend to express the complexity of the code. They are available through static code analysis.

Static software metrics have the ability to predict errors [8]. A great deal of the current software error predictors use static software metrics [7]. The reason for this is the high correlation of these metrics with errors and the availability of automatic software analyzers. The publication of software error datasets, such as the NASA Metrics Data Program and multiple other datasets [40] has also accelerated the research of using static software metrics to predict errors.

One of the most used metrics is cyclomatic complexity, proposed by McCabe [41]. Many of the public error datasets [40] use this metric, and it is produced by commercial static code analysis software[42, 43]. Watson and McCabe conclude that "the cyclomatic complexity measure correlates with errors in software modules" [44]. There are also some claims that cyclomatic complexity is not ideal for predicting errors. Some have found no correlation [38], while others argue that it is as good a predictor as code line amount [39]. Similarly to the coverage criteria, cyclomatic complexity uses con-

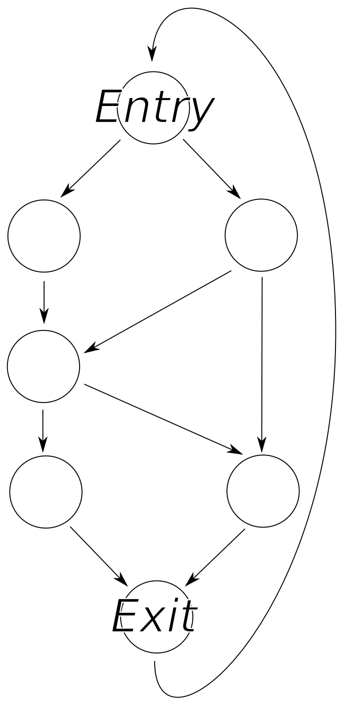


Figure 15: An example control flow graph, with exit point connected to entry point

trol flow graphs to describe the program. To form cyclomatic complexity, exit points are connected to the entry point. Cyclomatic complexity is then the amount of cycles that do not contain other cycles. This can be also calculated from Equation 2

$$Complexity = E - N + P \tag{2}$$

where E is the number of edges, N the number of nodes and P the number of connected components, which, if one program is in question, is always one. For the example graph in Figure 15, the complexity is $E - N + P = 10 - 7 + 1 = 4$

Halstead has also created a set of metrics that describe program complexity [45]. Both McCabe and Halstead metric suites describe the procedural nature of the source code, but metrics to measure object-oriented design also exist. Chidamber & Kemerer created a set of metrics to describe object-oriented programs [46]. It characterizes class features such as inheritance and method design.

In addition to the calculated metrics, basic development statistics such as the number of added and deleted source code lines over time have been used for feature data [47]. Simply the date can be an effective, although not very actionable, predictor of erroneous code change, which leads to the notion "Don't Program on Fridays" [48]. These basic statistics are effortless to extract as they only require for a software project to have an existing version control system.

Organizational features

Organizational features describe how software is developed in terms of organization structure. In an example development case, the authors of certain changes in a project are mapped to an organizational tree of the project. It is then determined on what level and in which team the change has been made. Some of the key organizational features are [36]:

- Distribution of work between teams
- Depth of the author in the organization tree
- Changes in the organization such as a developer leaving the project

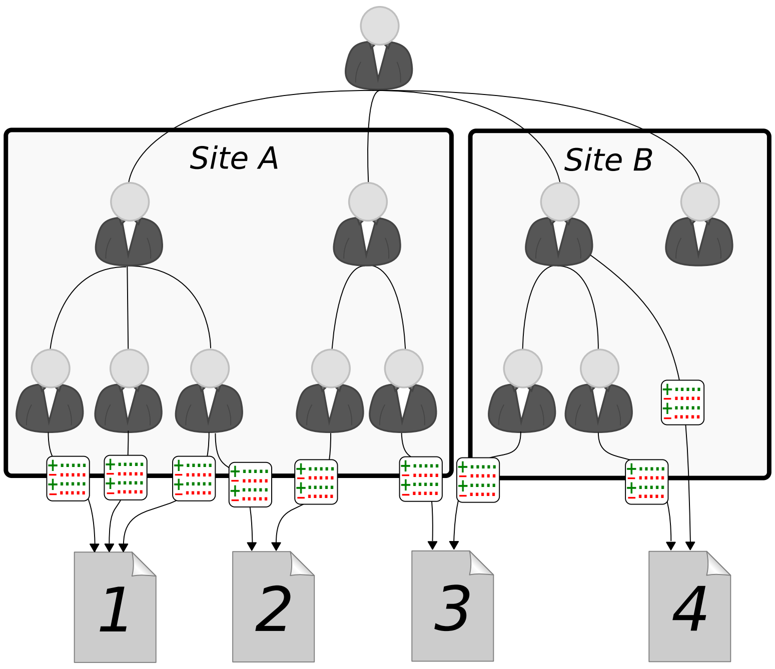


Figure 16: Organizational features

The files 1-4 in Figure 16 describe the different ways of dispersion between file ownership

1. Between individuals
2. Between teams
3. Between sites
4. Between different depths in organization

The use of organizational metrics is motivated by the idea that a significant part of errors originate from problems in project communication and distribution. A case study done at Microsoft found that the best predictor was organizational metrics [36]. It was better than more common metrics obtained from static code analysis.

Tracking developer activities

The most recent approach to feature data is the collection of activity history from a developer's local development environment [37, 49], also called micro interaction metrics. By recording what file changes a developer makes in a personal environment, a finer granularity can be achieved for file change data. Depending on the developer, a great amount of iteration that eventually amounts to the final commit can be absent from the version control history. The problem with these metrics is that they require intrusive tracking software and limit the developer to using a certain development environment.

2.3.2. Classifier data

Classifier data is an equally important part of machine learning. In the case of error prediction, classifier data is the classification of file changes between erroneous and non-erroneous code. The phases of error induction and fixing are shown in Figure 17. First a commit is made which induces an error. Later on the error is found and an error report is filed for the error. An error-fixing file change is made for the error, and in the end it is verified that the fix works. At that point, the error is said to be closed.

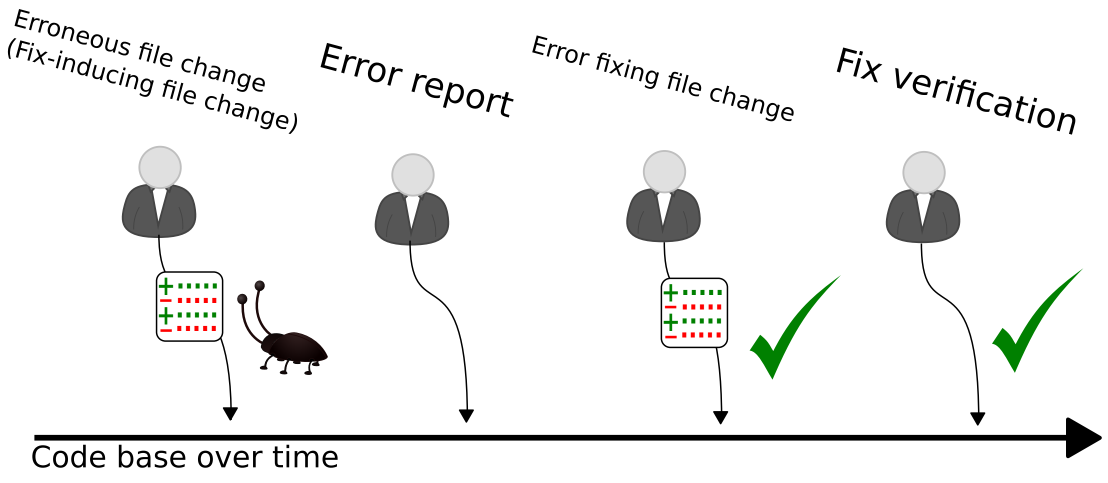


Figure 17: Error phases

Error-causing code

Error-causing code, sometimes referred to as fix-inducing code, is code that later on gets fixed by an error-fixing commit. It features a change in the code that causes an undesired function of the program logic. Error prediction relies on the data about what code was erroneous. The more accurate the data are, the better the prediction is.

Three different error reporting scenarios exist in typical projects:

- The code change that caused the error and also the fixing code for the error is reported.

- Only the fixing code is reported.
- Neither is reported.

In an ideal case for error prediction, erroneous code would be marked accordingly into an issue tracking system. That way the location of an error fix and the code that caused the error would be tightly coupled and ready to be used in analysis. In a worse case, the erroneous code does not get blamed. The worst possible case is that neither the erroneous nor the error-fixing commit are identified in the development system. The most usual case in projects that use an issue tracking system, is that the fix commits refer to an error ticket, so the fixing code gets reported.

In many open source and enterprise projects, the erroneous code does not get blamed. Most of the time even the error-fixing commits are not identified properly. The imperfect or missing information about the error-causing and fixing commits has motivated many researches on the subject of predicting what is fix-inducing code [50, 48, 51, 52] and error-fixing code [53] in the development. SZZ algorithm is one of the methods for detecting which code is error-fixing.

SZZ algorithm

It is possible to infer error-causing code from the code change history even if the information about which code changes caused the errors is missing. This is the case when the error-fixing code changes are identified from the rest of the code changes. The first published algorithm for searching fix-inducing code in a code base is the SZZ made by Śliwerski et al. [48] Since the publication of SZZ, many researchers have made their own adaptations of the SZZ algorithm, and it has become a standard for error prediction projects in their data-collection phase. See Appendix 10.1 for a detailed description of the SZZ algorithm.

Kim et al. improved on the original SZZ algorithm, which they claim to produce a considerable amount of false negatives and false positives [54]. Two of their main concerns are that the annotation information given by the SCM is insufficient, and the fact that the SZZ algorithm could not distinguish changes that did not amount to the functionality of a program. The same topics were also revisited in another study done by Williams and Spacco [50].

The first version of the SZZ algorithm is not able to distinguish changes that do not affect the way a program functions, but which are still relatively simple to prune. These changes comprise formatting, e.g. adding whitespaces and line breaks, and commenting. By ignoring these changes, 18%–25% of false positives and 13%–14% of false negatives were able to be removed in a case study by Kim et al. [54] Williams and Spacco used a coding-language-dependent tool to further remove changes that do not affect a program's functionality. Nevertheless, there exist code changes that are too hard to automatically label as non-behavior changes, which require deep static or dynamic analysis [54].

SZZ is known to come up with some false positives and negatives. Bird et al. [55] argue that bias in datasets for error prediction casts doubt on the effectiveness of error predictors that are taught with biased datasets.

2.3.3. Machine learning

Machine learning in general is the practice of fitting mathematical models and algorithms to predict or classify data on the basis of a given set of example input data. Many different approaches to machine learning, such as logistic regression and gradient boosting, exist. The most important factor affecting the performance of these models is the selection of the feature set.

Feature selection

Not all extracted features are useful. Some of the features might be redundant, too noisy or just not good predictors. A redundant feature correlates with existing features and does not improve a predictor. A noisy feature contains random variance, which in turn does not correlate with the actual parameter that is predicted. There is also a chance that a feature is not correlating with the classification, i.e. it does not tell anything about the underlying process.

Some of the features have to be left out to create a high quality predictor, for example to reduce overfitting of the model [56]. It is possible to use exhaustive search to search for a subset from a small feature set. The number of subsets for an n-sized set of features is:

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

Consequently, using exhaustive search is not possible for large feature sets because the number of subsets grows very fast.

Overfitting is a problem of machine learning models. When selecting features, a model will fit a finite set of sample data usually better when more features are added. But then the probability of overfitting the model will also rise. The Akaike information criterion [57] takes into account the number of parameters as a deteriorating factor for model goodness. The Akaike information criterion is shown in Equation 3. Amount of parameters in the model is described by k and L is the maximum likelihood of the model.

$$AIC = 2k - 2\ln(L) \tag{3}$$

Hill climbing algorithm can be used for searching an optimal solution to a problem. Hill climbing starts from an initial solution. Then it changes a single parameter and uses a heuristic to determine if the latest change results in a better solution to the problem. If so, the next change will happen incrementally over the latest change. The algorithm will continue until no more changes can be made to make the solution better.

Hill climbing can be used for feature selection if there is an error heuristic defined for the problem. Initially the feature set consists of one feature. Then either a feature is added from the set of available features, or, later on, a feature is removed. Hill climbing varies the parameters so that at every step of variation, if the error heuristic has gone down, the new variation is deemed good and further variations are built on top of that variation. However, if the number of errors has risen, the variation is reversed. Hill climbing has the possibility of finding only a local maximum. This means that if there are multiple maxima to be achieved, hill climbing will not necessarily find the best feature set.

Simulated annealing builds upon the hill climbing algorithm. It reduces the chance of landing on to a local maximum by adding a certain amount of noise into a variation step. In other words, there is a chance that the search does not go towards the direction that seems at that point to be the better one. At the start of the search, variance of the noise is at the highest. This means that the search will cover a larger area of the feature set space by randomly jumping through the search space. Gradually, the variance of the noise is diminished and the chance of being near a global maximum presumably rises. This makes simulated annealing work more like a hill climbing algorithm towards the end of the search. At the end, a simulated annealing search will settle on to a maximum that has a bigger chance of being the global maximum than in a hill climbing case.

Logistic regression

Logistic regression is a relatively simple machine learning technique. Logistic regression is shown in Equation 4, where P is the probability of error, β the variable coefficients[1] and X the measured variables. The logistic function is useful for predicting probabilities as the input variables can range from positive to negative infinity while the output will stay in the range of zero and one [58].

$$P = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots)}} \tag{4}$$

Logistic regression has been successfully used as a machine learning technique to predict the overall chance of software failure [59] and locations of errors in the program code [35].

Boosting

Boosting is a machine learning technique that aims to reduce the errors caused by the bias of the data. This is done by applying many weak learners instead of one strong learner. Weak learners learn to classify a subset of the data but do not perform well on the whole data set. AdaBoost is a highly acclaimed [60] boosting technique introduced in 1995 by Freund and Schapire [61].

Boosting works by iteratively fitting a model to a problem. At each stage, another weak learner is introduced to compensate the shortcomings of the previous weak learners. At the end, the multiple weak learners will converge into a strong learner. [62]

Figure 18 shows an example [62] of how boosting is used to solve the XOR problem. In the problem, four samples exist in the sample space:

$$z = f(x_1, x_2)$$

$$z_1 = -1 = f(0, 1); z_2 = +1 = f(-1, 0); z_3 = +1 = f(1, 0); z_4 = -1 = f(0, -1)$$

which are to be classified by the predictor. The weak learner functions h_1, h_2 and h_3 are iteratively chosen, so that the subsequent function will compensate for the error

[1] β_0 often does not have a meaningful interpretation in the model, but without it, the model would be biased.

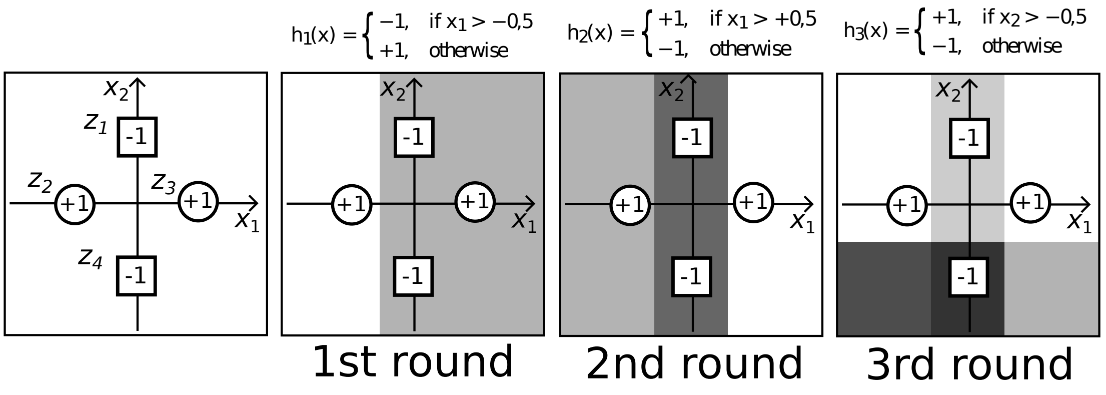

h1(x) = { −1, if x1 > −0,5; +1, otherwise
h2(x) = { +1, if x1 > +0,5; −1, otherwise
h3(x) = { +1, if x2 > −0,5; −1, otherwise
x2
x1
z1
z2
z3
z4
-1
+1
1st round
2nd round
3rd round

Figure 18: Boosting used for the XOR problem

made by the previous function. The function is then summed in the ensemble function with a proper weight ρ.

For example at the first round, function h_1 is chosen. Function h_1 is a predictor that tells that every sample that has $x_1 > -0{,}5$, is classified to have the value -1. This function correctly classifies z_1, z_2 and z_4 but misclassifies z_3, hence more weak predictors' functions are needed.

The process of adding weak predictors continues until the final model can then correctly classify the samples. Thus it is shown that the final model can classify the samples, although the functions h_1, h_2 and h_3 by themselves cannot correctly classify the sample space. The final formula for classification is shown in Equation 5.

$$H(x) = \sum_t \rho_t h_t(x) \tag{5}$$

Gradient boosting is a boosting technique described first by Jerome H. Friedman [63]. It uses a group of weak predictors, such as decision trees, for prediction. In gradient boosting, the "shortcomings" are defined as negative gradients [64]. The combination of using regression trees with gradient boosting is called gradient tree boosting [65].

2.3.4. Predictive testing

Predictive testing is a novel approach to regression testing introduced in this thesis. Predictive testing works by prioritizing test cases which are estimated to find the most errors. If the testing time is limited and the tests take too long to execute, only the most important tests can be run. The prioritization speeds up the testing process, because the errors are found earlier. In Figure 19, the basic concept of test case prioritization is explained. When there is a limited time frame, only a part of the whole test case mass can be executed. In this case, executing the test cases randomly will result in inefficient testing. When tests are ordered by the probability of them finding an error, testing will be more efficient as the errors are found in the limited time frame.

The given example of only executing a part of the test mass is not applicable for final product verification. In that case the motivation to ship a flawless product leads to the execution of every test case. In the case of continuous integration, test execution time is limited by the need of having a fast feedback loop for the developers. Other factors, such as expensive and limited testing resources, can also limit the testing time that can be given to an iteration of the product in a continuous integration scheme.

Previous regression test prioritization methods have involved prioritization by changed binaries [66] and by code coverage [67].

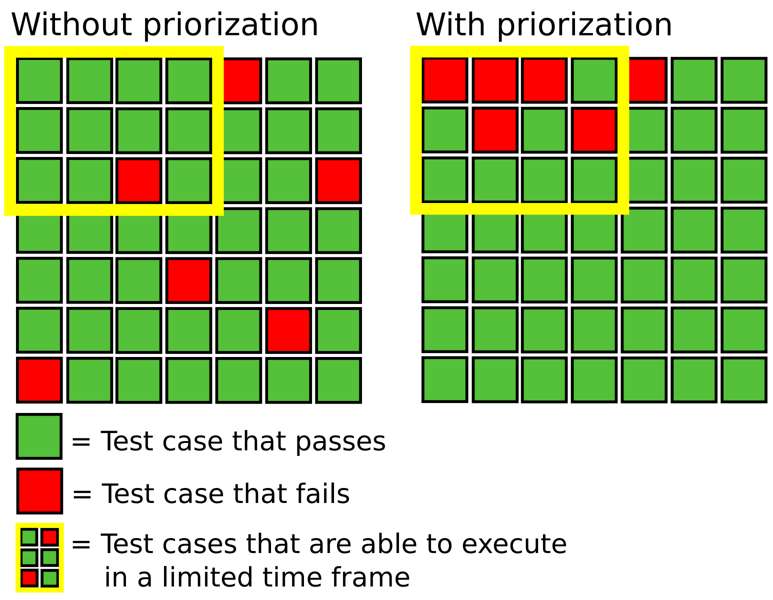


Figure 19: Test case prioritization

3. PROBLEM : RESOURCE EFFICIENCY IN TESTING

The predictive implementation was created for a software project at Bittium (formerly known as Elektrobit). The software project in question was required to produce certain features and it was part of a very large infrastructure. The produced software had several services that were produced in several software layers. The product also had real time requirements. The characteristics of the project cause the software to become large and complex. The complexity and large size of the software project causes issues in the following areas:

- Need of comprehensive verification
- Challenges in communication
- Resourcing and optimizing resource usage

These issues are not present in a smaller, less complex projects. For example, the following characteristics describe PC software with a client-server architecture:

- Software components run in the same environment
- Software components can be implemented, verified and tested with the same mechanisms
- The server can effectively be stubbed or mocked while testing the client and vice versa

$\rightarrow$ I.e. very few extra and expensive resources are needed

However, these characteristics do not describe a project that, for example, is implemented as an embedded system which has to conform to the standards of a large communications infrastructure. Hardware creates some challenges, as software has to work in different cores such as:

- The main embedded multicore CPU, which is unlike the development environment, e.g. an ARM processor instead of a x86 architecture processor.
- Digital signal processor
- FPGA or ASIC

In such a network project, integration testing is complex as the product needs to be verified against user equipment and different infrastructures such as control and billing. These require expensive equipment for testing. The amount of testing is also vast as these kinds of projects tend to have a large amount of non-functional requirements to fulfill, such as accessibility and robustness. The difference between a simple project and a complex one is illustrated in Figure 20.

To verify that the produced software conforms to all of its requirements, rigorous testing is needed. On the other hand, to test complex software, an equally complex testing system is needed. Testing equipment in a large project is expensive, as is the time spent testing.

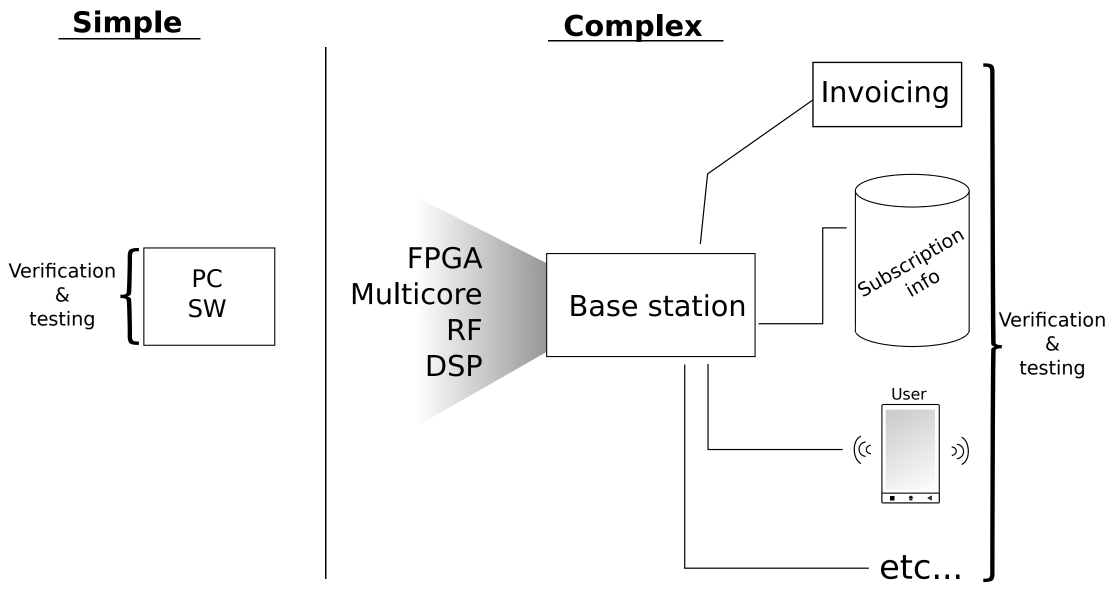


Figure 20: Example of a simple and a complex software project

The communication between developers is not immediate in a project that has a large number of developers in multiple developer teams. The problem of communication becomes even larger when development is done in separate geographical locations. In this situation, reports and queries made by developers have to pass multiple layers. The responsibilities of each individual can become tangled, and eventually managing the software development process becomes challenging.

The complexity of the software causes loss of visibility into the software. For example, pinpointing the root cause of a problem found in verification becomes harder when the project becomes bigger. The visibility to the work done by an individual also diminishes by the factor of complexity. This also creates challenges for managing the software process.

All of these factors eventually spawn errors that cause the software to function unintendedly. The errors then cause loss of profit that is proportional to the phase where the error is found. Circumventing the creation of these errors is not conceivable, but finding them as early as possible reduces the cost associated with them.

These problems can be tackled with the help of data that are gathered from the development process. The project had development tools that could be queried for data from the whole timespan of the project. This data can be used for teaching predictors that can give data-driven decisions. The data mined can also be used in their own right to produce visibility that could not otherwise exist. The data can be used to give solutions in areas such as:

- Targeting testing effort
- Targeting code reviews
- Visibility to the software process

Regression tests need to be run for every new version of the software to make sure that the existing functionality has not been broken. Running every possible test for a version is very expensive and increases the time needed to release. Executing just a certain part of the tests decreases the time needed to run the tests, but then tests that would find an error could be missed.

The time taken in testing can be reduced and the effectiveness can be maintained. This can be done by testing only the most error prone parts of the software. This reduces the time while keeping the number of errors the tests can find the same. This maximizes the value of testing.

Code reviews are one of the main communication channels between developers. Code reviewing is an effective tool for maintaining quality. However, it is also costly, as it demands time from experts that otherwise would be implementing new features. Similarly to the case of targeting testing, targeting the efforts of code reviewing would maximize its value.

The information that is stored in the different development tools tells what happens in the software project. The problem is that this information is distributed between the tools. To use these data in the management process, they have to be processed to give meaningful insights of the development process. The pre-processed data can then give valuable information for management purposes. The same information can also be used for understanding where the communication issues happen.

4. DESCRIPTION OF SOLUTION

The main output of this thesis is a predictor program. The predictor is piloted at Bittium in a software component project that is part of a mission-critical embedded systems project. The software component has around 100,000 lines of code and is implemented in the C++ language. The development for the software component is done in two geographical locations and has about 60 developers. It has four main development teams and a separate testing team.

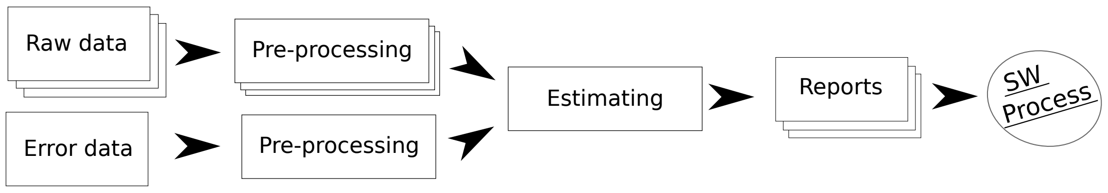


Figure 21: Predictor

The predictor is described in Figure 21. The predictor program is an automated process. It is designed to run as a timed service on a remote server. At first, the program extracts the feature information from its external data sources. The data are then pre-processed, and meaningful links inside the data are made. The pre-processed data are used for the learning process of the predictive model. Finally, the predictor outputs different reports about the project. A more detailed figure of the predictor is shown in Figure 22. It shows the inputs and outputs and the intermediate steps of the predictor.

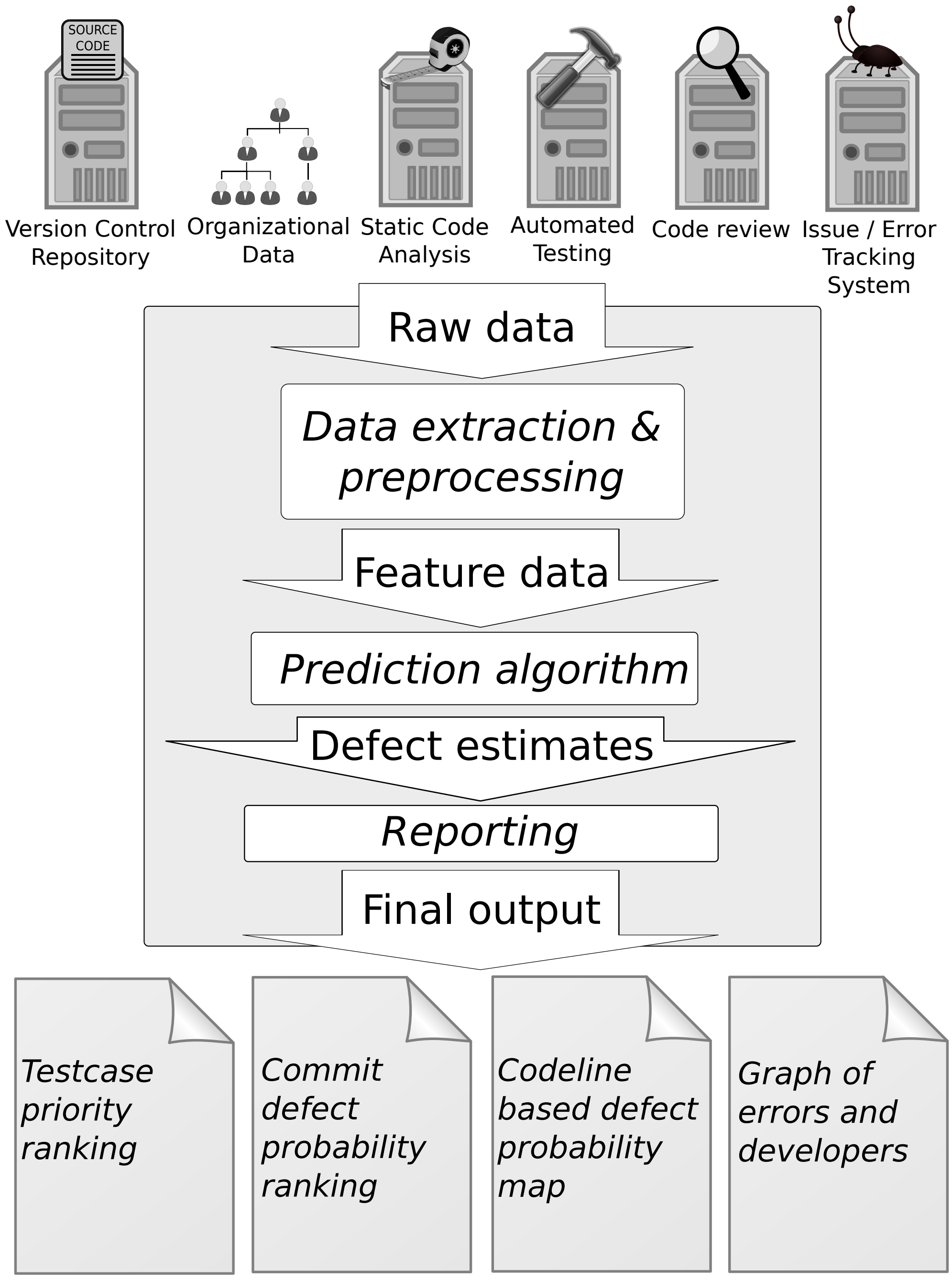

SOURCE CODE
Version Control Repository
Organizational Data
Static Code Analysis
Automated Testing
Code review
Issue / Error Tracking System
Raw data
Data extraction & preprocessing
Feature data
Prediction algorithm
Defect estimates
Reporting
Final output
Testcase priority ranking
Commit defect probability ranking
Codeline based defect probability map
Graph of errors and developers

Figure 22: Overview of the resulting predictor

The system implemented had to be automated. This meant that the gathering of raw data and error data, pre-processing, estimation of the errors and reporting were automated. This enables the system to report as frequently as it is required. The reports were chosen to be done daily.

The predictor's data are processed before creating reports, so viewing the reports does not require further interaction with the source data. This means that when the reports are created once a day, the predictor could run for a whole day if there are not any other constraints such as other programs in need of processing time. This sets a loose performance requirement for the run time of the predictor. Low performance requirements allowed the predictor's development to be concentrated on the overall quality of the predictions and the reports.

The predictor was implemented in Python. Python made it possible to create a predictor prototype fast. This, in turn, made it possible to review the results and steer the prediction model into the correct direction from the start of the predictor project. Python is suited for fast prototyping as it is a high-level programming language with a high availability of free open source libraries.

4.1. Feature extraction: raw data & pre-processing

Feature extraction was the first step needed for creating the prediction. A commit is the basic element of software development. It is also the starting point for feature extraction. The different sources for feature data are tied together by the information in a commit. The relationships between commits and other feature data are shown in Figure 23.

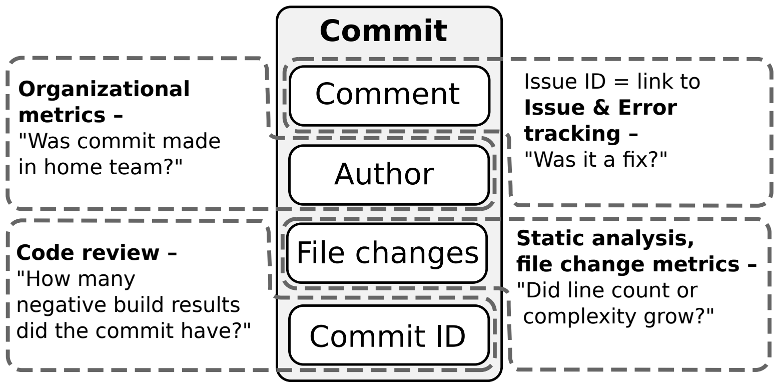

Commit
Comment
Author
File changes
Commit ID
Organizational metrics –
"Was commit made in home team?"
Issue ID = link to Issue & Error tracking –
"Was it a fix?"
Code review –
"How many negative build results did the commit have?"
Static analysis, file change metrics –
"Did line count or complexity grow?"

Figure 23: Links between commit data fields and other data sources

Feature extraction involves communicating with the external systems using the APIs provided by them. It collected the historical data spanning from the start of the project to the present day, a period of approximately two years. At the end, feature extraction produces feature data that can then be used for prediction. An overall view of the feature extraction process is shown in Figure 24.

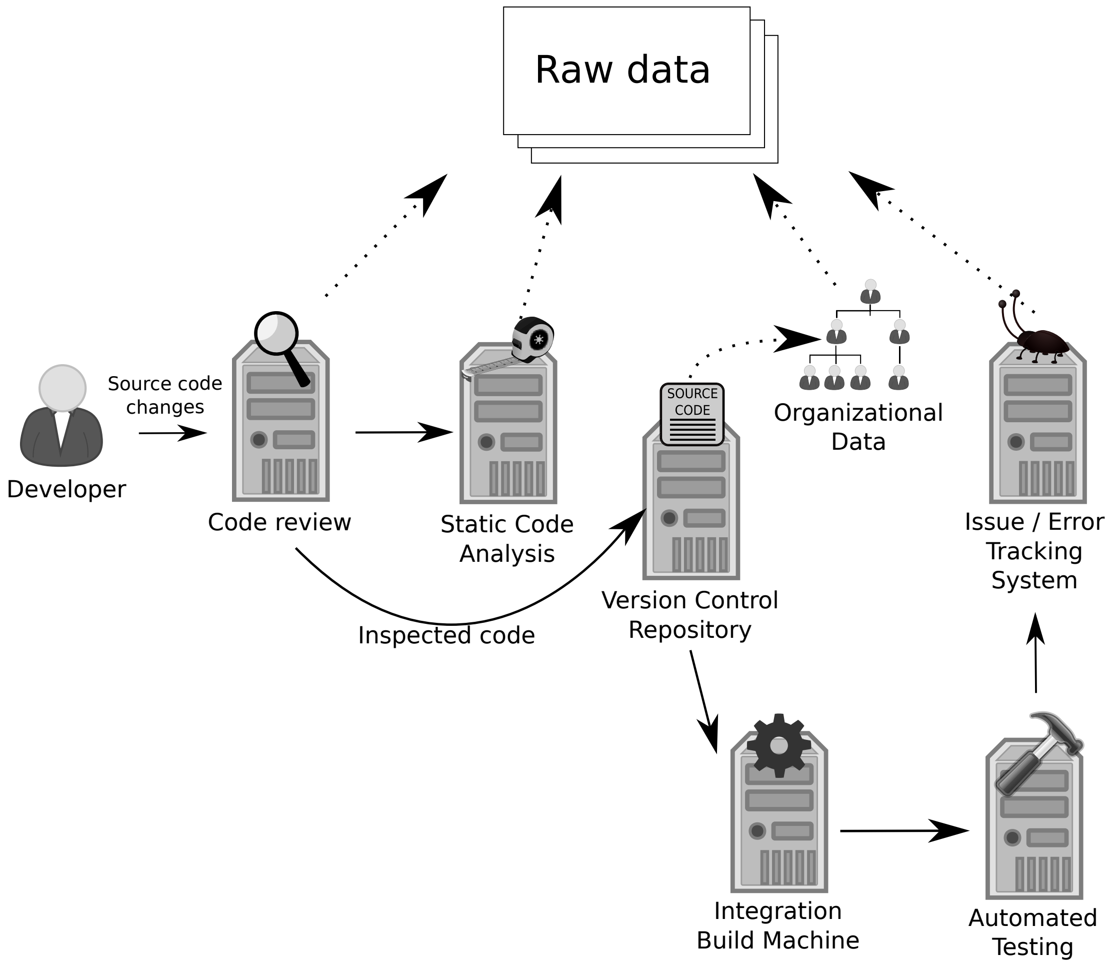


Figure 24: The valuable collection points of development data

Features were extracted for every file change inside commits in the commit history. This means that some of the features that describe the commit were duplicated over the file changes. This is illustrated as an example of the resulting feature data in Table 3.

The lists containing extracted features are marked with a ❖ in this chapter.

Table 3: Example of the resulting feature data table

Author	# added lines	# deleted lines	# files changed	*Commit ID*[a]	*Filename*
John	23	5	3	*16ca91249*	*foo.cpp*
John	2	15	3	*16ca91249*	*bar.cpp*
John	59	42	3	*16ca91249*	*baz.cpp*
Jane	1	2	2	*c805b94a8*	*foo.cpp*
Jane	38	45	2	*c805b94a8*	*bar.cpp*

[a]The commit ID and filename are included only in the example — they are not useful predictor features

4.1.1. Version control history

Version control history was the starting point for data mining. The full history of commits was extracted from the version control system as a commit log. Some of the features were readily available in the commit log (✓), but most of them had to be processed from the commit data (✗). The commit log was parsed for the following features:

- ❖ The name of the commit's author (✓)
- ❖ The number of files changed (✓)
- ❖ The number of added and deleted lines (✓)
- ❖ The subcomponent's name (✗)
- ❖ The total number of commits made into the file (✗)
- ❖ The total number of edits by the committing developer (✗)
- ❖ The number of subcomponents changed (✗)

The project had its subcomponents separated in the directory structure by a certain pattern. This known pattern was used to infer the subcomponent that was changed in a certain commit. The subcomponent name was used for calculating the amount of different subcomponents changed in one commit. The total number of commits by the committing developer to the file and the total amount of commits by every developer to the file are calculated as a sum of the previous history of commits to the file.

In addition to these variables, the data from the commit log were used for connecting the commits to the data from other systems.

4.1.2. Static software metrics

Cyclomatic complexity was chosen as the main complexity measure. The choice was made on the basis of ease of implementation and past knowledge of cyclomatic complexity's ability to describe code complexity.

At first, a proprietary static code analyzer was used for calculating cyclomatic complexity. Soon it became clear that the complexity information was needed for the whole history of the commits. This was possible by going through every version of the software between the commits in the version control. This meant that the cyclomatic complexity calculations would have to be done for a large number of commits.

Static analysis using the proprietary tool was too extensive and time-consuming when done for every build in the history. The proprietary tools were replaced by ccm [68]. By feeding ccm with only the changed files for every commit, the whole commit history was analyzed in just two hours.

Cyclomatic complexities were calculated for functions but the predictor estimates error-probabilities for file changes. Therefore the per-function cyclomatic complexities needed to be mapped into per-file change measures. The aggregation functions that were used are:

- The average CC of the functions in the file
- The weighted average CC of the functions in the file. Weighting is done by the source line count of the function
- The maximum CC of the functions in the file
- The minimum CC of the functions in the file

For each of the aggregated measures, the difference that the file change made is calculated. This is done by subtracting the previous measure of the file prior to the file change from the current measure. The difference is calculated, because the delta caused by a file change predicts the errors that are done in that file change alone.

In addition to complexity measures, the delta of source lines and the count of functions were extracted.

4.1.3. Peer review and build history

The software project had a peer review system and a build system. Together, their target was to make sure only commits of good quality would be integrated into the product. The build was made on every commit and the commit was tested against unit tests. The features that were extracted from these systems were:

- The length of code review

- The number of negative and positive outcomes from:
  - Code reviews
  - Builds
  - Unit tests
- The reviewer being the same as the committer

It was noted that sometimes the commits in the peer review system were reviewed by the committer. Some of these cases were explained by a random malfunction in the build or test system that forced the commit to be re-reviewed. Some of them did not have an explanation, so this self-approving feature was decided to be left in the extracted features.

4.1.4. Organizational metrics

Organizational information was extracted from a database of employees. From there, the location of the developers was extracted in addition to the information about who worked under whose management. From the management information, it was possible to determine which team the developer belonged to. Based on this, the following features were extracted:

- The number of different sites and teams where the file has been changed
- The percentage of changes made to the file at the home site and by the home team
- Was the change made at the home site and by the home team?
- The number of unique developers that had edited the file
- Had the developer edited the file before?

The home team and site was determined for a file to be the team or site where most of the past file changes had originated from.

4.2. Error data

Error data tells which of the commits are error-causing. The classifier data was collected by inspecting the commits that were fixing the errors. The fix information for the commits was gathered from an issue tracking system.

4.2.1. Raw error data: Issue tracking

Issue information was stored for the project in an issue tracking system. This system also had the errors of the projects.

From the start of the project, developers had been instructed to include the ID of the added feature implementation or the ID of the error fix into commit comments in the version control. This made it possible to find error fix commits by searching the commit comments for the issue IDs. To find out the error fixing commits, the issue tracking system was queried with the issue IDs found. The issue tracking system then gave information about the affected software version, the name of the discoverer and the date of discovery.

Some of the commits had multiple issues listed. This was remedied by multiplying the file change data over the multiple issues. This was done similarly as in the case of multiplying commit data over file changes, as shown in Table 3 on page 39.

4.2.2. Error data pre-processing: Estimation of fix-inducing code

The errors in the project did not have information about which commits or file changes had caused them. The error-causing commits were extracted with an algorithm that resembles the SZZ algorithm. There were three main revisions made of the algorithm and all of them used the error fixing commits to find the commits that caused those errors.

The first version of the algorithm searches the commit history for the commit that most resembles the error fixing commit. Resemblance in this case is defined as the number of edits in the same files. The most similar commit is then marked as erroneous.

The second version of the algorithm takes the file changes made in the error fix under inspection. It finds the last commit that edited the file into the form that it was before it was fixed. This is done by searching the commit history of the edited file with a code snippet that tells the form of the code before it was fixed. The code snippet is formed by taking out the added lines and inserting back the deleted lines of the error fix commit. When every file change of the error fix commit has been searched for, the commit that had the biggest number of hits — i.e. file changes that had edited the files to the form they were before the error fix — is deemed to be error-causing.

The third version of the algorithm is similar to the second. Instead of choosing one commit to be erroneous based on the amount of hits, the third version blames every file change that is found to be a hit, without caring about commits per se. In addition to this change, the information about the software version that is affected by the error is used. If an error is perceived in a certain software version, then it can be said that the error must have been done before the release of this software. This sets the last possible date for the erroneous file changes, and if a hit is found later than that date, it can be said to be a false positive.

Figure 25 shows an example of how these three algorithms find the cause of the error ERR-590. Every commit shown in the figure have file changes that completely rewrite one function inside the file. The three algorithms determine that the cause of the error ERR-590 is:

1. Commit 2 and all the file changes it has. Commit 2 has the largest number of file changes to the same files as the error fix commit.

2. Commit 3 and all the file changes it has. Commit 3 has the largest number of file changes that have edited the lines to the form that they are when they are fixed by commit 4.

3. File changes to files B and D in commit 3 and a file change to file A in commit 2. The change in file C in commit 1 is determined not to have caused the error, because the commit has been done before the release of version 3.17, which is the affected version of the error ERR-590.

The first and the second version of the algorithm operate on commit level. When a commit is deemed to be error-causing, all of the file changes inside the commit are classified as error-causing. This means that they classify several extra, non-erroneous, file changes as error-causing.

The third algorithm, however, had the problem of creating too much dispersion in the classifications. Because of the realities of the software project, some of the commits that were identified to be error-fixing by the committer also had file changes that were not actually error fixes. The error-fixing commits would, for example, include refactoring and, in some cases, new implementation. However, the third algorithm tries to find a "suspect" file change from the history for every file change in the error-fixing commit. This means that a great deal of false positives will be created in the error data set. To overcome this, the second algorithm is chosen for the error predictor. This is justified because a commit represents a unit in the development process. On average, the file changes are more connected between each other in a commit than file changes between different, separate commits.

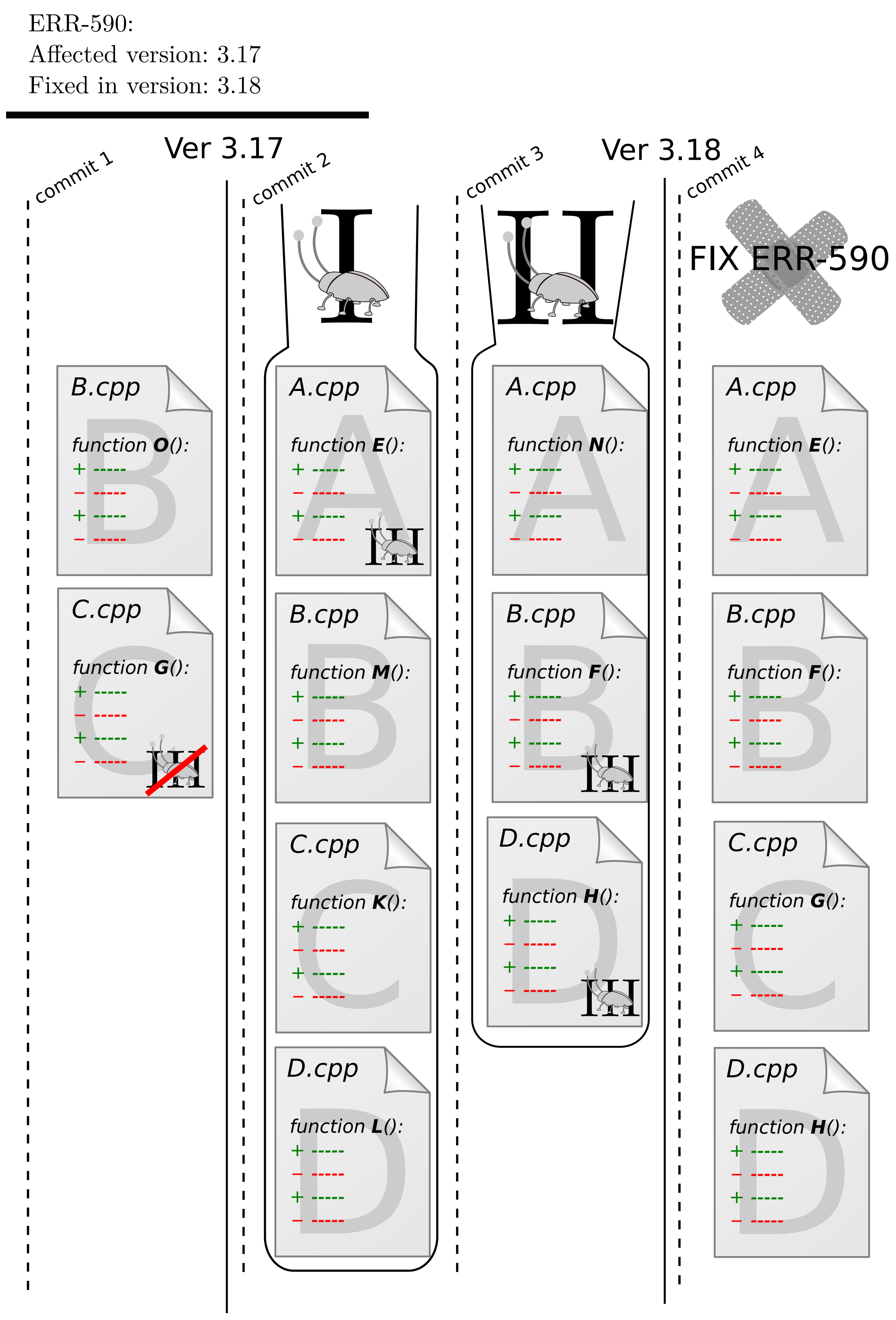


Figure 25: Estimation of fix-inducing code done by the three different algorithms. Note how the change in function G in commit 1 is not blamed for the error by algorithm III because the change has happened before version 3.17.

4.3. Statistical model

The predictor's capability of predicting errors is based on a machine learner's ability to predict errors. A method was created to choose the best possible machine learner for this case. The predictor was tested each month by reviewing how its predictions compared to the known history of the recently completed month. The evaluation method followed these steps:

1. Use the historical data for training until the current month ($T_0 \ldots T_{N-1}$).
2. Rank the completed month's (T_N) file changes by the error probability given by the predictor.
3. Plot the percentage of detected errors (Figure 26) when going through the ranked code changes. The effort for one file change is proportional to the number of lines of code it has changed or added.
4. Based on the rankings, analyze how much effort is required to detect a certain percentage of errors.

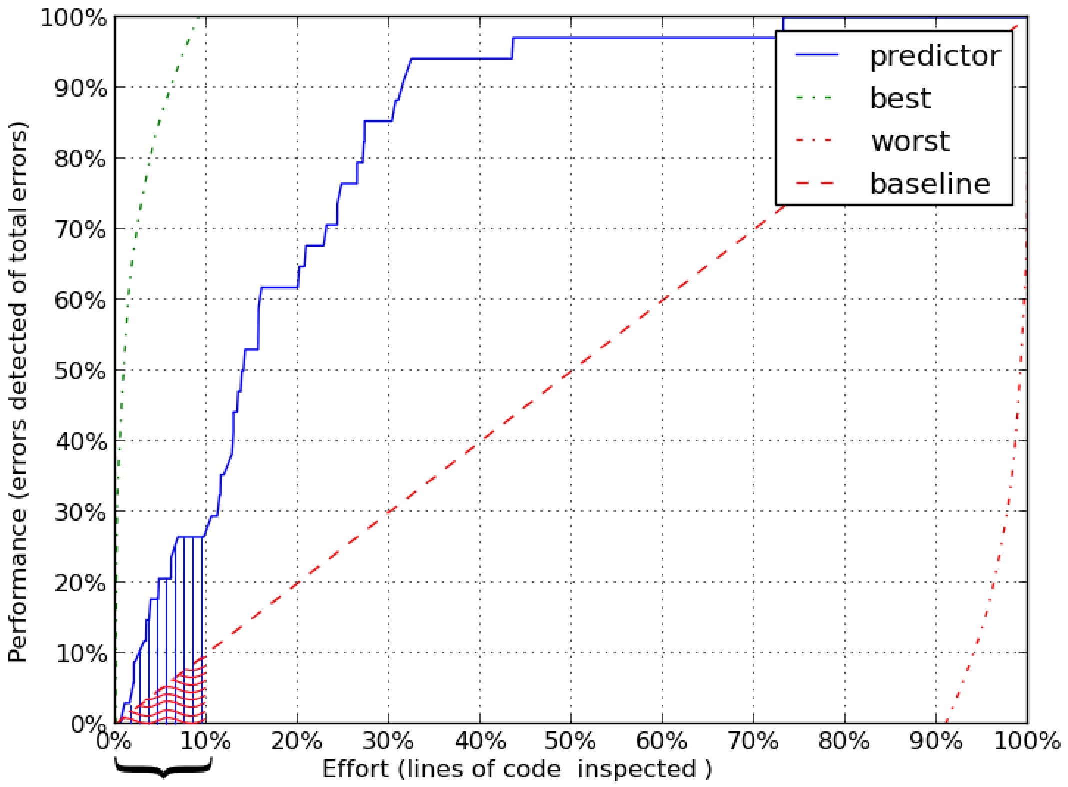


Figure 26: Performance graph

$$Performance = \frac{\int FoundErrors}{\int Baseline} \tag{6}$$

A numerical value of performance was needed later for the model selection. The ratio of the area under the found errors graph and the area under the baseline was used, described in Equation 6. However, it was noted that sometimes it is useful to maximize a fast growth of found errors and not care about the whole accuracy of the model. This would be the case, for example, when the scenario is to go through the most (estimated) risk prone 10% of the code base. Then the performance is the relation between the first tenth of the area under the found errors graph and the first tenth of the area under the baseline. In Figure 26, the area under the error graph is marked with vertical blue lines and the area under the base line is marked with horizontal red waves.

4.3.1. Model selection

The Python library scikit-learn[65] offered various machine learning algorithms. Of said algorithms, logistic regression and gradient tree boosting were chosen to be evaluated. Eventually, choosing the better model was done by evaluating the different models with the method described earlier.

In the end, it was concluded that logistic regression suited error estimation better. Gradient tree boosting was prone to overfit to the training set, which led to unnecessarily complex models and nonsensical predictions. Logistic regression is also justified by the fact that different features in the data are, by themselves, directly proportional to the probability of an error.

Logistic regression had the added benefit of simplicity. Since logistic regression essentially estimates the coefficients for the model features, the coefficients can provide information about how each feature affects the probability of errors in the file changes. This information was later used for displaying the reasons why a commit was marked as being suspected of having an error.

4.3.2. Feature selection

Not all of the extracted features were useful. Some of them were either redundant or irrelevant. Feature selection was used for picking the features that predict the errors best. The feature sets were evaluated using the evaluation method described earlier.

As the number of all of the features is 28, an exhaustive search would have to go through $2^{28} \approx 3 * 10^8$ subsets. Searching through every combination would simply take too long, so a different method had to be used.

The first solution was to make use of an expert's opinion to reduce the amount of features before an exhaustive search. An exhaustive search was then made using 13 chosen features and the evaluation took two days to complete.

The second solution was to use a hill climbing algorithm that promptly evolved into a simulated annealing algorithm. Simulated annealing had the possibility to choose from the whole set of features. In addition to the feature set, simulated annealing was used for choosing the optimal parameters for the different machine learning algorithms.

Figure 27 shows one simulated annealing search that had 4,000 iterations. It chose features for a logistic regression model from a set of 28 features. It also varied the model's parameters. The green line is the performance of the current set of features.

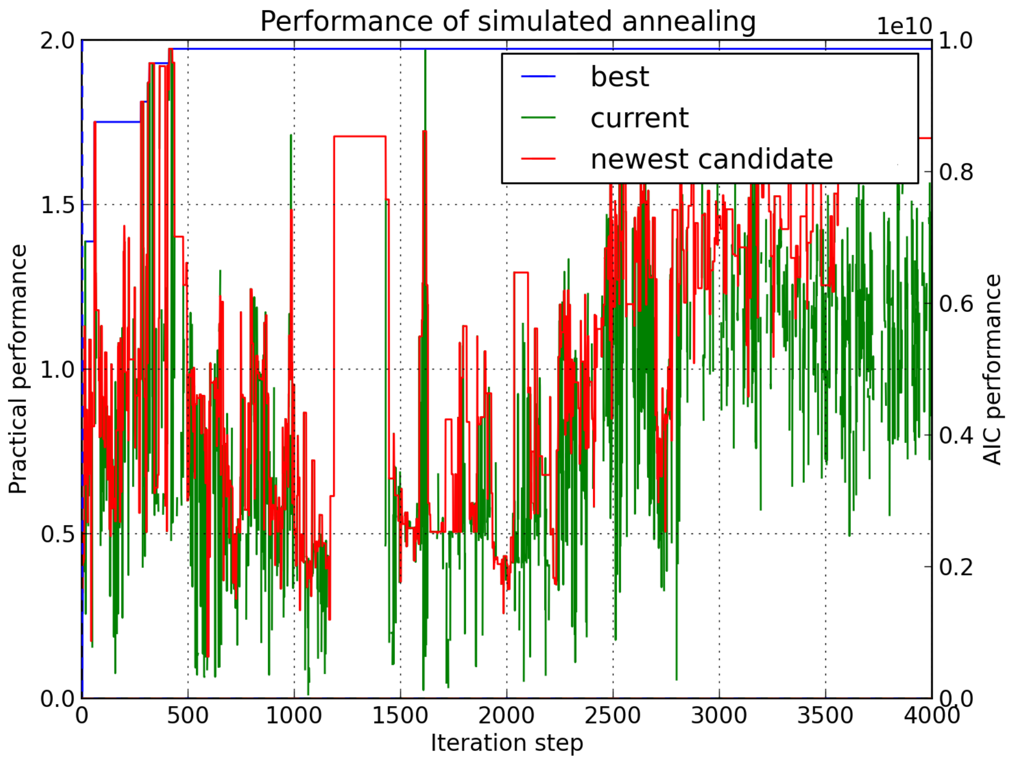


Figure 27: Model selection using simulated annealing

The red line shows the performance for the latest candidate varied from the current candidate. At first, the latest varied candidate is chosen almost every time. As the search continues, the latest varied candidate is only chosen when it performs better than the current candidate. The blue line is the latest candidate found best during the search. It does not affect the simulated annealing search.

A single iteration in this kind of selection scenario is either varying the model parameters or the features. The model features are varied by either adding new features to or removing existing features from the feature set. The graph shows that single steps in model selection can have a significant effect on the model's performance. Furthermore, it can be seen that the candidate found best was found moderately early in the search. At the end of the search, the performance goes up for the model but still does not reach the performance of the earlier, best feature set.

In this case, the steps are so big for the simulated annealing search that it is relatively impractical for searching for the optimal feature set. Additionally, the features that were eventually chosen by the simulated annealing algorithm were the same that were found to be the best by the expert-aided exhaustive search.

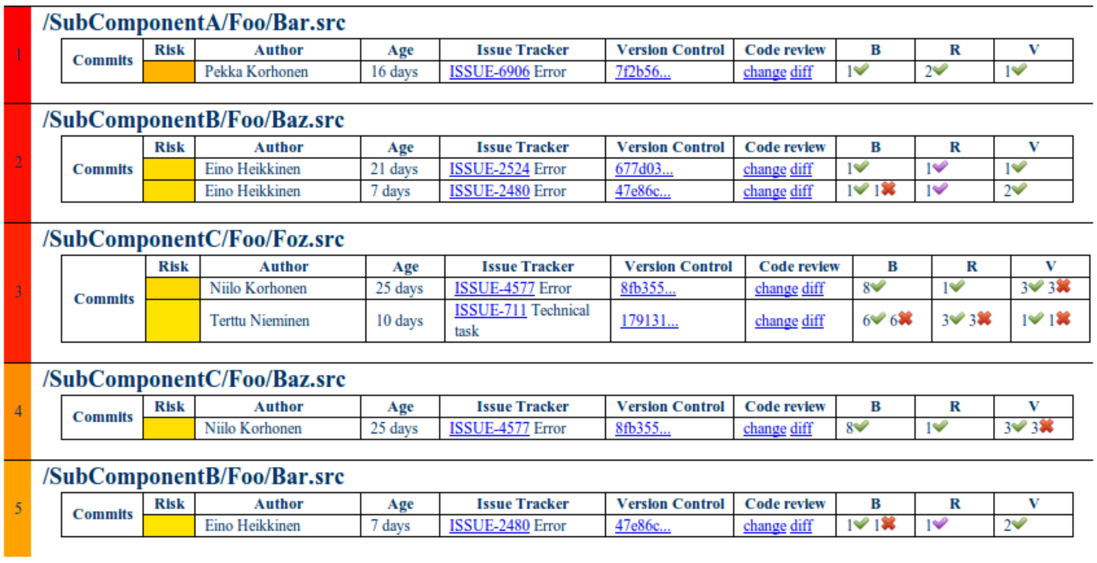
4.4. Reporting

The predictor gave a prediction of how probable it is that a file change causes an error. That by itself is not very useful for developers working with the code, nor is it very useful for the continuous integration system. The prediction data for file change error probabilities was used to create:

- File change hotness ranking
- Code hotness graph
- Error-developer graph

4.4.1. File change hotness ranking

Figure 28 shows the file change hotness rankin. It lists the file changes that are estimated to be the most likely to cause an error. The file change ranking was limited to ranking the commits that were made in the past 30 days. The ranking also had meaningful links to tools that were used in the development process. The issue ID in commit comments led to the issue in the issue tracking system. The commit ID led to the gatekeeping and peer review system, where it was possible to quickly view the changes made in the commit. The listing also shows passed and failed builds, reviews and verifications for all of the commits. In Figure 28 a passed outcome is signified with a green check mark, while a failed outcome is denoted with a red cross mark. The special case where the commiter and the reviewer were the same person was shown with a purple check mark.



1 /SubComponentA/Foo/Bar.src

	Risk	Author	Age	Issue Tracker	Version Control	Code review	B	R	V
Commits	🟧	Pekka Korhonen	16 days	ISSUE-6906 Error	7f2b56...	change diff	1✓	2✓	1✓

2 /SubComponentB/Foo/Baz.src

	Risk	Author	Age	Issue Tracker	Version Control	Code review	B	R	V
Commits	🟨	Eino Heikkinen	21 days	ISSUE-2524 Error	677d03...	change diff	1✓	1✓	1✓
	🟨	Eino Heikkinen	7 days	ISSUE-2480 Error	47e86c...	change diff	1✓ 1✗	1✓	2✓

3 /SubComponentC/Foo/Foz.src

	Risk	Author	Age	Issue Tracker	Version Control	Code review	B	R	V
Commits	🟨	Niilo Korhonen	25 days	ISSUE-4577 Error	8fb355...	change diff	8✓	1✓	3✓ 3✗
	🟨	Terttu Nieminen	10 days	ISSUE-711 Technical task	179131...	change diff	6✓ 6✗	3✓ 3✗	1✓ 1✗

4 /SubComponentC/Foo/Baz.src

	Risk	Author	Age	Issue Tracker	Version Control	Code review	B	R	V
Commits	🟨	Niilo Korhonen	25 days	ISSUE-4577 Error	8fb355...	change diff	8✓	1✓	3✓ 3✗

5 /SubComponentB/Foo/Bar.src

	Risk	Author	Age	Issue Tracker	Version Control	Code review	B	R	V
Commits	🟨	Eino Heikkinen	7 days	ISSUE-2480 Error	47e86c...	change diff	1✓ 1✗	1✓	2✓

Figure 28: File change hotness ranking of one month

File change hotness ranking gives a view into the development process for the person managing it. Depending on the size of the project, the number of the commits that are merged into mainline can be large. Some of the commits can be more unstable than others and need more testing. By seeing which commits are the riskiest and making sure that they are thoroughly tested, a manager can direct the efforts to improve the overall quality of the software.

Commit hotness ranking was tested in one development team. The leader and his team were instructed to watch the ranking for one month. The commit ranking was perceived to show the actually most error prone commits. The team leader said that the information was already known to him and his team, but he still did not deem the ranking useless.

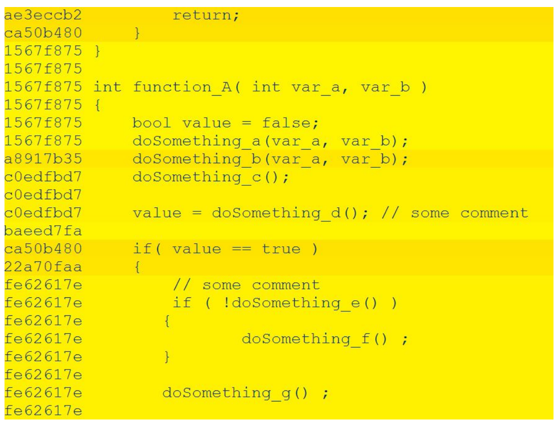
The team also hoped hotness ranking would give information as to why the commits were estimated to be error prone. A version that showed the features that had the strongest effect on the decision was also made, but it was still said to be stating the obvious.

4.4.2. Code hotness graph

The code hotness graph, a tool for viewing the current status of the software project, was also made. The basic idea of code hotness is that every line of code is marked with the error-proneness of the commit that has created the line. This tool then showed the aggregate chance of error for a file. It also showed the aggregate chance of error recursively for the whole directory structure. The chance of error for a file P is calculated from the chance of error in lines $p(i)$ with the formula shown in Equation 7.

$$P = 1 - \prod_{i=1}^{N} 1 - p(i) \tag{7}$$

Figure 29 shows the code hotness graph for a part of source code file. Every line is colored on the basis of its probability of causing an error, yellow indicating a low probability and orange a more higher probability.



```
ae3eccb2            return;
ca50b480        }
1567f875 }
1567f875
1567f875 int function_A( int var_a, var_b )
1567f875 {
1567f875     bool value = false;
1567f875     doSomething_a(var_a, var_b);
a8917b35     doSomething_b(var_a, var_b);
c0edfbd7     doSomething_c();
c0edfbd7
c0edfbd7     value = doSomething_d(); // some comment
baeed7fa
ca50b480     if( value == true )
22a70faa     {
fe62617e         // some comment
fe62617e         if ( !doSomething_e() )
fe62617e        {
fe62617e                doSomething_f() ;
fe62617e        }
fe62617e
fe62617e        doSomething_g() ;
fe62617e
```

Figure 29: Code hotness graph

4.4.3. Error-developer graph

Figure 30 shows a graph that was made to study and visualize the interaction between teams and software errors. The creation of this graph was possible using the data that were gathered in the predictor's data gathering phase. The graph was part of the automatically generated report suite.

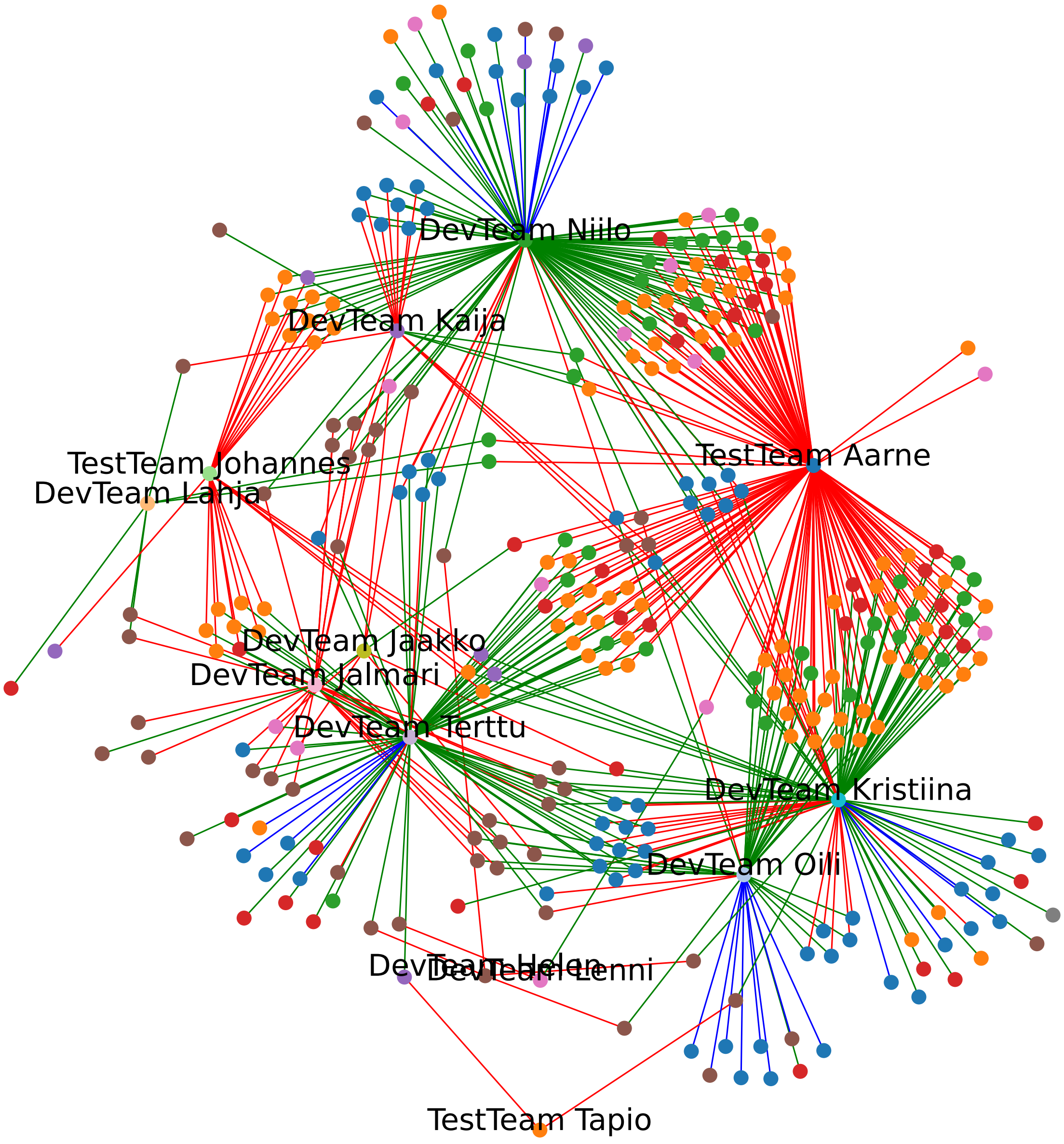


Figure 30: Graph of errors and teams[1]

In the error-developer graph in Figure 30, the nodes that have a name over them are the teams. The teams that are labelled as "DevTeam" are developer teams, and the teams that have the label "TestTeam" are teams concentrated on testing, such as integration and verification teams. The nodes without a label are the error nodes. The

[1]For confidentiality reasons, the graph contains names that are anonymized, the true amount of errors is not shown and testing levels are left without names.

error nodes are colored on the basis of the testing level where the error was found. The connection is colored in

Red between an error and the team which reported the error

Green between an error and the team that fixed the error

Blue between an error and the team that both reported and fixed the error

The graph also had the option to change the red links to describe the creator of the error, and, in turn, the blue links to describe links to the errors that are created and fixed by the team. It was also possible to change the graph to show errors between developers instead of teams. The graph in Figure 31 shows errors between developers, with links that are color coded using the same principle as earlier.

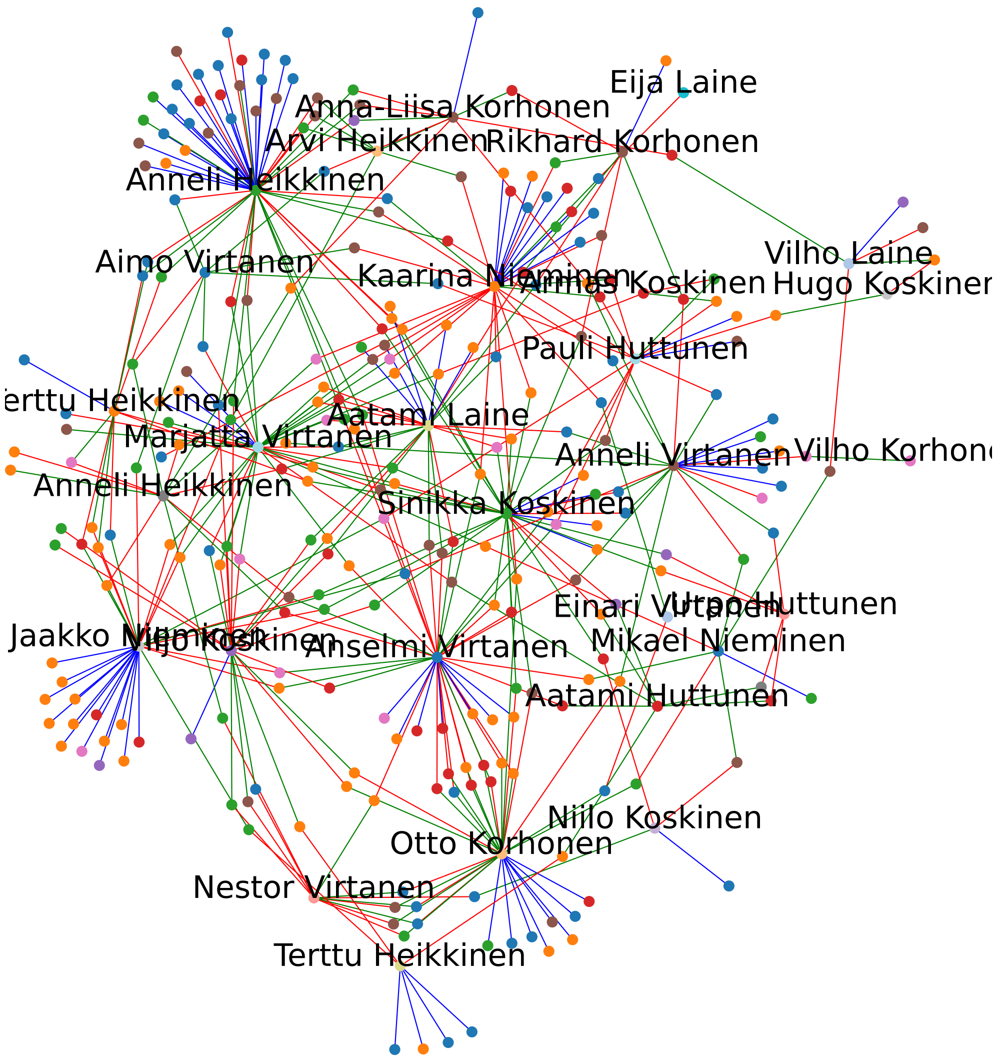


Figure 31: Graph of errors and developers[2]

[2]For confidentiality reasons, the graph contains names that are anonymized, the true amount of errors is not shown and testing levels are left without names.

4.5. Test case prioritization

Test case prioritization of system test cases was one of the initially planned applications of the error predictor. To map the predictions with the test cases, additional mapping of test cases and the source code is needed. Since the predictions were made on file change level, test cases would need information of which exact files they are testing. The project did not have information about which test case would test which part of the code. This is understandable, as this kind of information would require additional effort that would have been carried out from the start of the project. One such mechanism is code instrumentation.

A method was invented to alleviate the problem of missing code and test case mapping. The error data had information about which test case found the error. The fix that was created for the error in turn had information about which lines were changed in which files. As the errors are found by testing and fixed by the developers, a mapping between test cases and code would eventually emerge. This is demonstrated in Figure 32.

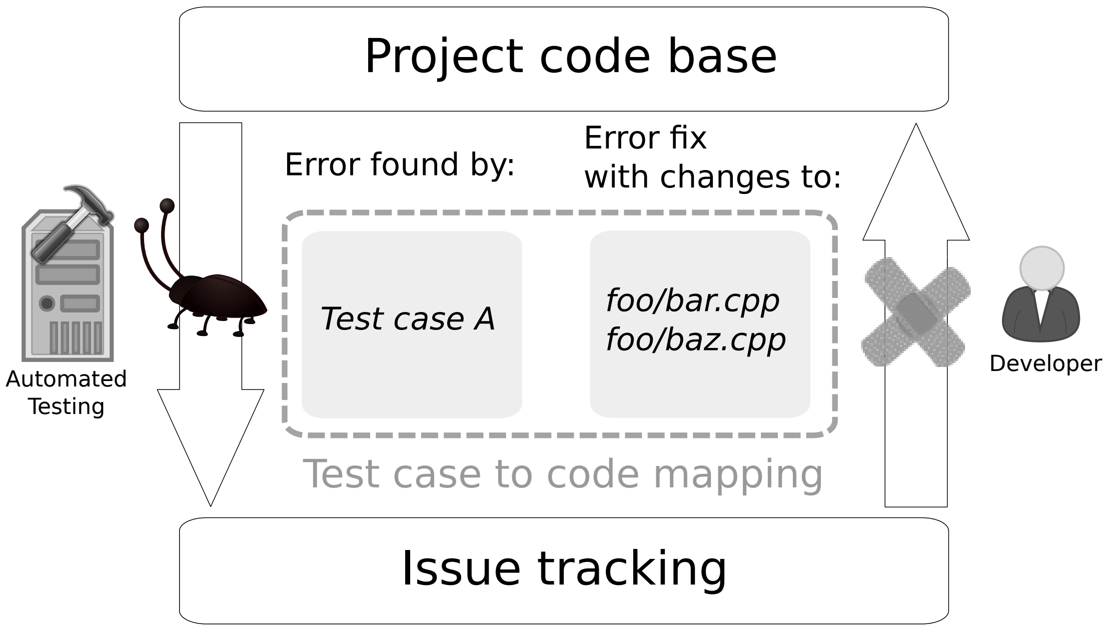


Figure 32: Example of a method which maps test cases to code

It turned out that this method resulted in too vague a mapping. Although there were a lot of errors found by automated system level tests, the amount of errors was too small to create a mapping between test cases and program code.

5. ACHIEVEMENTS

The goal of this thesis was to improve the software quality of a big software project by using error prediction. One of the steps towards that goal was to point out the most probable error locations. When the most probable error locations are known, corrective action can be taken early and therefore the cost of these corrective measures stays low.

The initial step for creating the predictor was to get data about where errors are created. An error blaming algorithm was created, and it was able to mark the locations where the errors were created. The error blamer was verified during the development to roughly mark the correct error-causing file changes.

Performance tests show that the predictor could rank the changes based on their probability of creating an error. The ordering the predictor made prioritized the file changes that were later found to be actually erroneous. In the case of file change hotness ranking, the predictor was perceived to predict the errors correctly. This tells us that the predictor, along with the error blamer, works not only in theory but also in practice.

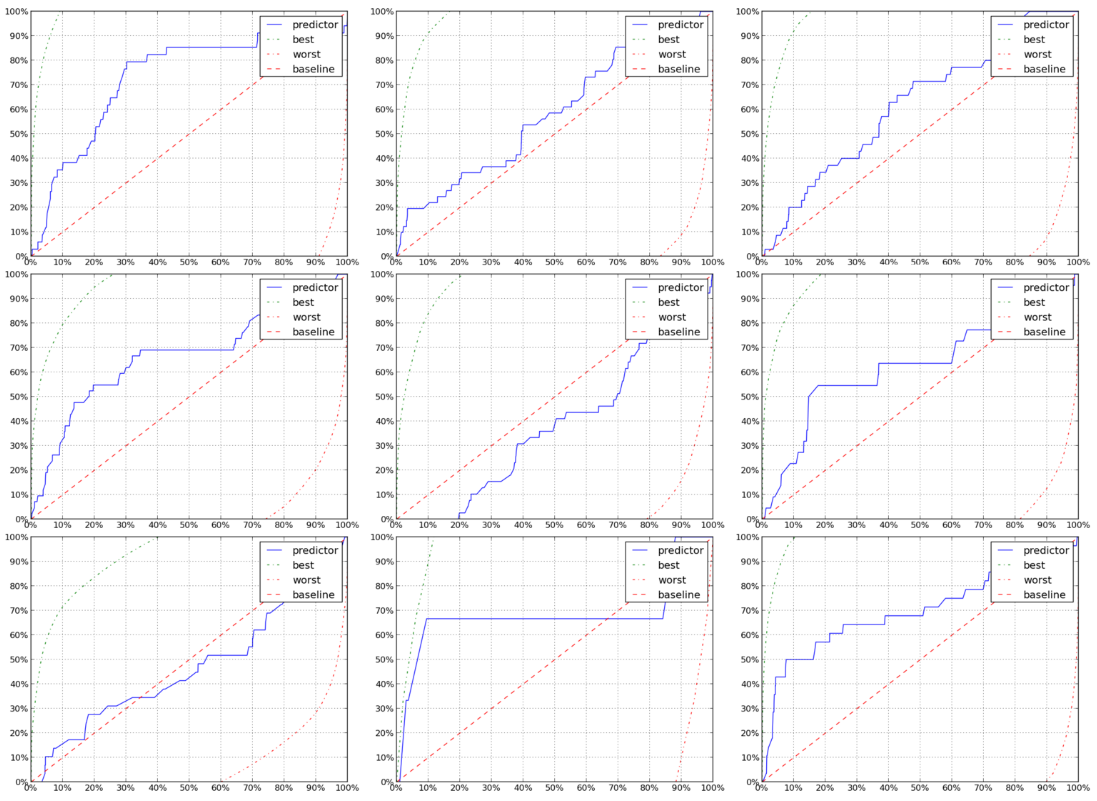


Figure 33: Performance of the predictor over nine consecutive months

Figure 33 shows the predictor's performance over nine consecutive months. On average for these months, the error predictor sorts over a quarter of the erroneous file changes into the first 10% part of the file changes. This means that, for example, if the time for code reviewing is very limited, going through the 10% of the highest ranked code changes is 2.5 times more effective than going through the changes randomly.

Another goal for the predictor was to improve testing efficiency by prioritizing the test runs. Although the predictor was capable of predicting the error locations, a ranking of test cases could not be achieved. This was because of missing data that would map which test case would test which part of code.

The original goal of prioritizing the test cases using the prediction data could not be achieved. Still, other improvements to software quality were accomplished through using the same data. The predictor gave understanding of the most problematic practices in the software development process.

A broader goal for the error predictor was that it would give answers to data-driven decisions. There was extensive work done for the gathering part of the error predictor data. The same work for gathering data was then used for creating an error graph. The error graph gave insights into the error data which were perceived to be very intriguing by the developers.

The code-hotness graph gives information about which parts of the software project are the most error-prone. This information can be used for targeting future code quality efforts, such as refactoring. Furthermore, the commit hotness ranking can give a "heads-up" for integrators and team leaders on particularly error prone commits. This information can be used, for example, in the prioritization of code reviews or for selecting the best reviewers.

5.1. Error graph

Figure 34 shows the error graph with circles around the error clusters. The circles with solid borders are around the error clusters that have been found by testing teams. The circles with dashed borders are around the errors that were found by a development team and had been fixed by another development team.

It can be seen from the error graph that most of the errors are reported by the testing teams, which makes sense as it is their job to find errors. The interesting clusters of errors are the ones between development teams. Those errors have been caught before system testing, but have still been found outside the team that has made them. In other words, they could have been found before integrating them into the mainline. Based on this, it can be argued that by improving the internal integration testing of the teams, the costs caused by spreading the error to outside teams could be cut. This would require efforts in creating a communication channel between the teams that would enable effective intrateam integration testing. This could be in the form of anything ranging from documentation to shared software between the teams.

5.2. Publications

Part of the results presented in this thesis have also been published at the 2013 International Symposium on Empirical Software Engineering and Measurement [69]. The error graph described in this thesis has also been featured in detail in a workshop article by B. Turhan and K. Kuutti [70].

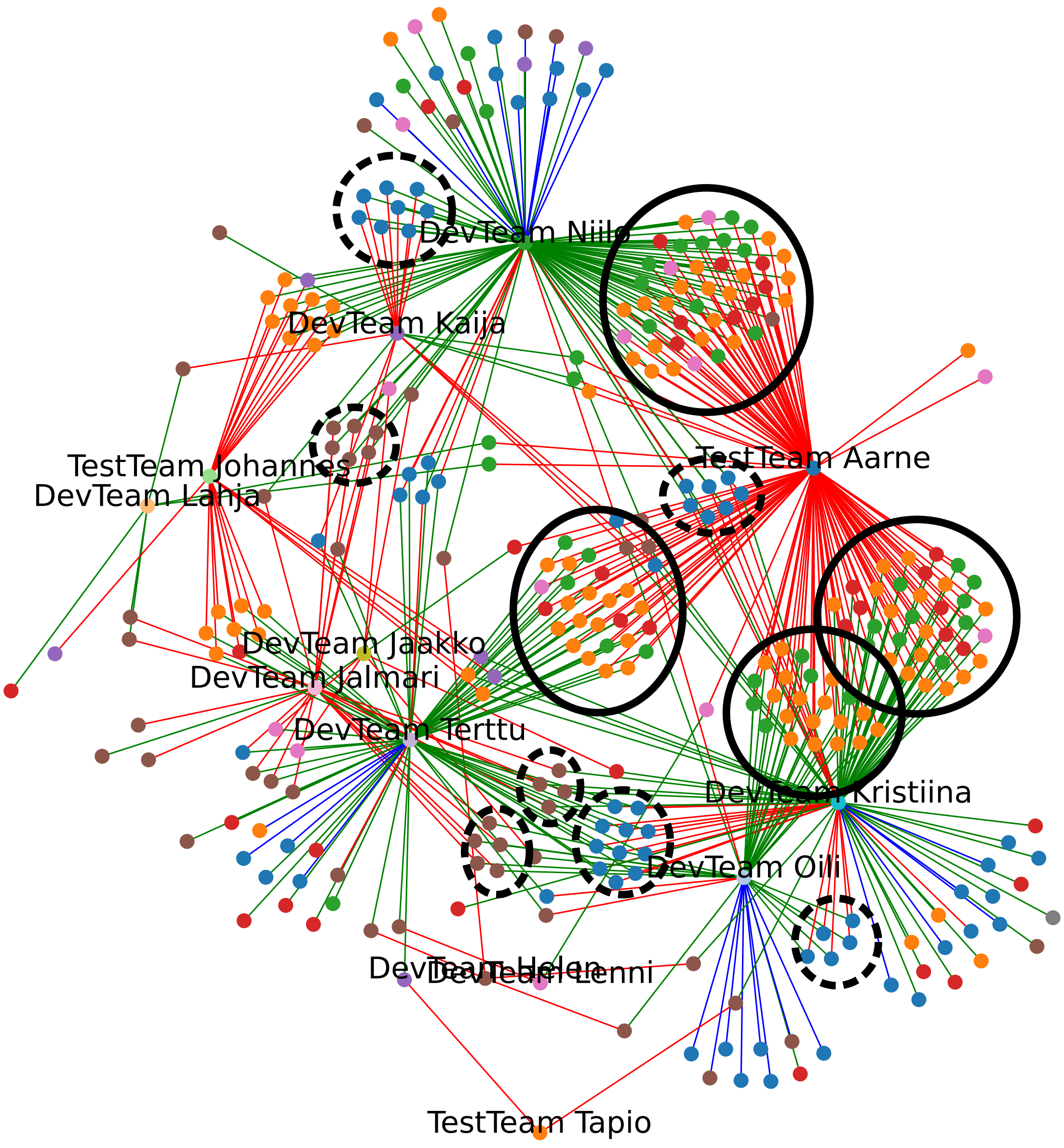


Figure 34: Error graph with circles around the error clusters

6. FUTURE WORK

The future work for the error predictor includes improving the feature data. Test prioritization was not achieved, but some improvements could be made to enable it in the future. The error predictor could also be used for improving efficiency of continuous integration in ways other than test case prioritization.

6.1. Improvements in feature data

During the development of the data extraction, some future improvements were proposed, such as using unit testing as feature data for the predictor. The predictor would also benefit from further filtering in the code change data.

6.1.1. Unit testing as a feature

Historical data from unit testing could have been an important source of feature data for the predictor. The problem with unit testing in the project was that, although there was unit testing, it was not possible to gather the historical data from it. There was not a defined way for preparing the environment and running tests that would have been valid for the whole project history.

The project should commit from the start to abstract all the necessary steps to run unit tests under one runnable task. In addition to being beneficial from a quality perspective, the commit history could be rewound and statistics about the unit testing coverage could be gathered. It would be especially enlightening to compare how changes in the code are taken into account in unit testing. Feature data that can be gathered from unit testing could also be a very useful predictor, as testing is tightly related to quality.

6.1.2. Filtering of code changes

One of the biggest causes of noise for the predictor data is the blaming process for the errors. While there were multiple versions of the error blamer which were improving one after another, the final version still had room for improvement. For example, while comparing the differences between versions of source code, the blamer did not use the semantics of the programming language in question. This caused erroneous blames, because the blaming process would occasionally blame errors on code changes that would change whitespaces or comment sections. These changes could not possibly be the actual source of the error.

A solution for this would be to compare the differences between revisions without whitespaces and comments. To further improve accuracy, the different revisions could be compared after pre-processing or even after compilation. A change that does not affect the final compiled byte code cannot be the source of an error. These improvements would, in turn, cause the data extraction phase to be significantly slower.

6.2. Test case prioritization

To prioritize test cases, a mapping between them and the source code was needed. Creating this mapping was problematic. The mapping was done using only the knowledge of which test case found an error, and what was changed in order to fix that error. This method could have been improved by expanding the mapping by using the understanding of how the software worked. One rule that could be used to improve the coverage data would be: "If a certain code line is executed by a test case, then the whole execution path before and after the line is executed by the same test case."

The mapping between tests and code could also have been done by using the issue tracking data. In the project, features are mapped to requirements, and some of the requirements are mapped to test cases. A big part of the commits were tied to a feature by using a feature ID. This means that if

- Code C implements feature F
- Feature F implements requirement R
- Requirement R is tested by test case T

then code C can be said to be tested by test case T. The accuracy of this mapping is dependent on how accurate the mapping is between test cases and requirements.

6.3. Error prediction in continuous integration

The main complaint about the error predictor was that the predictions were too obvious for the developers. The developers felt that they would not benefit from learning what code is predicted to be error-prone. From this complaint, it can be deduced that the predictions should not be given to the developers as they are, but rather the information should be used for something that could indirectly benefit the developers' daily work. Even though the information of which commits are most probably erroneous might be painfully obvious to the developers, this information is still relevant and most often only known to the developers. Non-developers, such as managers, could gain value from the error prediction data.

Another example use of the predictions could be the fine tuning of continuous integration efforts. In continuous integration, there is a need for fast feedback for the developers. Testing equipment, however, is limited and expensive. One possible way to make continuous integration faster is to include multiple commits in one integration step. This has the drawback that when there is an error, the commit that caused it is not known. In this case, continuous integration has to go back and rerun the tests separately to know which of the commits caused the error. Normally, the following recursive algorithm would be used to find the error:

1. Go to the halfway point of the commit history, decide if the error still exists
2. If it does, take the first half of the commit history under inspection and go to step 1

3. If it does not, take the latter half of the commit history under inspection and go to step 1

In this case, the error causing commit could be found faster if the probability of causing an error was predicted for the commits. By bisecting the commit mass at the point where the cumulative sum of probability of causing an error is halved, the process can be faster.

7. DISCUSSION

There were challenges faced when constructing the predictor. There was also a possible threat to validity that is discussed. Some of the main results of this thesis are the notions made in good and bad practices in software development. Finally, the lessons learned while making the predictor are discussed.

7.1. Challenges in data

Some problems were faced while gathering and processing the raw data. Some of the commits consisted of changes with differing motives: for example, they had refactoring in addition to bug fixing. This suggested that some of the commits were too large and should have been split into a set of smaller commits. Another problem was that the commit comments could have either no or multiple issue IDs listed. The commits that had no issue ID but were error-fixing were ultimately missed by the predictor. However, commits with multiple issue IDs would confuse the predictor. In general, the commit comments did not always properly describe the meaning of the commit.

While the challenges in the data were a problem when developing the predictor, it was beneficial for the project to note the existence of these problems. As they were noted, the reporting procedures were updated. For example, a template was made for the commit comments. These kind of improvements support future data mining activities, but also benefit the project by having stricter and more readable commits and commit comments.

The validity and strictness of the data will always be a problem for automated data gathering. In this case, it would not even be feasible to require the developers to write machine-readable commit comments. After all, the provided version control tools exist to help the developers themselves. Finding a degree of strictness that both supports data mining and is valuable for a developer would be the most reasonable solution.

7.1.1. Obvious output data

A manager of a developer team was presented with the commit hotness ranking. He thought it did show valid information but that the information was already known to the developers. That is why this kind of tool could be useful for a situation where a manager has to control a process that spans multiple teams. In such a case, the manager would not be able to know what goes on in the software project without going through the commits.

One version of the commit hotness ranking showed the factors that contributed the hotness the most. This had the same response as earlier: it showed information that the team already knew. The factors, like the fact that the commits were big, were already known to the team.

7.2. Threats to validity

Missing thorough validation of the error blaming model might be a threat to the validity of the results. The performance of the error predictor is statistically measured using the blaming data. If the blaming data is incorrect, the performance value could be misleading.

The error blaming data does not have to be perfectly accurate for the error predictor to work. It is enough if the blamed error data correlates with the actual error data, since then the error predictor can be used for predicting errors when enough data are available for training the predictor. Some of the methods used for the error blaming algorithm guarantee enough accuracy that the estimated error commits correlate with the actual erroneous commits. One method, for example, is considering only the changes that have happened before the affected version. It should also be noted that only the file changes made at the same location in the same file are considered as possible candidates.

The predicted error locations were similar to what developers thought to be the most likely to have an error. This practical validation also affirms that the error blaming algorithm is blaming the actual errors.

7.3. Good and bad practices in software development

The predictor was based on logistic regression, which works by calculating the weights for different features. These weights can reveal good and bad practices in software development. The factors that correlated the most with the probability of the code having an error were:

- The number of lines of code changed in a commit
- Self-reviewed commits and the number of negative code reviews and build outcomes
- The distribution of commits between developers in different teams and at different sites

It was found out that a bigger commit was more problematic than a smaller one, i.e. the error probability correlated with the number of added lines. This was also true even if the predicted error probability was made proportional to the size of the commit. This further suggests that commits should be split into smaller entities in order to increase the overall quality and maintainability of the code.

The feature data gathered from the code reviews also showed significant correlation with error probability. The strongest correlation was with code that was committed and reviewed by the same person. This suggests that reviewing done properly can, in fact, increase the quality of the software. It should also be noted that the more negative code reviews a commit had, the more likely it was to cause an error. Here it should be taken into account that all commits that were inspected were ultimately accepted. Thus the ones that had negative reviews and failed builds were patched to finally pass the reviews and builds. It is hard to draw a conclusion of what this would mean in terms of practices, other than that the commit had gathered attention early on, maybe because it was a big one. In hindsight, it can be said that these bad code changes should not have been allowed. The fact that negative code reviews correlate with the probability of an error also suggests that code reviewing is an effective way to improve code quality.

Distribution of the commits from different teams and sites showed correlation with errors. As described earlier, a home team for a file is the team that has done the majority of the commits for the file. From the data, it was discovered that a commit not made in the home team was more likely to cause an error than a commit made in the home team. The same principle applied to geographical locations.

These notions that were found out in the project have to be considered with some caution. Although there was plenty of data that was processed, it only consisted of one project. This means that in some part the findings may describe the practices that are in use in the project in question rather than the practices that are in use in software development in general.

7.4. Lessons learned

Test case prioritization through error prediction is an advanced technique. This notion became clear in the process of creating the error predictor. Some of the problems when creating the error predictor originated from inconsistencies in the data. Missing information that could map the test cases to the code that they tested was the biggest challenge. However, if there are underlying challenges in the software development process, coming up with solutions to them can benefit the process more than an advanced technique such as test case prioritization. Before taking such a technique into use, the project should first look to benefit from easier approaches to development methods:

- The use of comprehensive unit testing
- Enforced code reviews by peers
- Well-formed commits, expressing only a single concept at a time
- Commits being tied to an issue tracking system

Unit testing is the double-entry bookkeeping system for the developer. The need for unit testing rises as the amount of collaboration and the size of the project grows. Having a locally runnable unit test set is especially useful for an embedded systems project, as the feedback for code changes would be slow otherwise. The project had unit testing. It did not have a well-defined way to run all unit tests from the beginning of the project. For example, a unit test make target is recommended, as it is needed if code coverage statistics are extracted over the project history. A defined way of running the tests is also needed for collaboration, as it tells every developer, and the possible continuous integration system, which criteria should be met at every point of development.

Code review done by a peer should be done for every commit. The prediction data showed that if a code review was only done by the committing developer, the probability of having an error was bigger. One way to better enforce reviewing is to use automatic reviewer recommendations. These recommendations can be made, for example, on the basis of who last modified the lines before they were changed in the commit.

High amounts of effort should be put into the structure of the commits. It was noted that high line count of added and removed lines inside a commit correlates with the chance of an error. This was true even if the probability was adjusted to the line count. From the data presented in this thesis, it can be said that commits should be small. There are measures that can be taken to make commits smaller. The software has to be structured in a certain way. Software traits such as loose coupling and high cohesion help when trying to create small commits.

However, there will always be a need for big commits. For example, a class is interfaced the same way in multiple different classes. To improve this, the functionality is refactored to be a method of the class. Then a project-wide refactor is done and loose coupling is introduced where it previously did not exist. In this case, splitting the refactor to multiple commits would require extra effort which would be made only for the sake of keeping the commits small. Therefore, in this case, trying to keep the commits small may not be justified.

An even more important factor than trying to keep the commits small is to have them to only produce one concept at a time. In the project, there were commits that implemented multiple concepts at a time, such as fixing errors and implementing new features, while even doing some refactoring. These kinds of commits tend to be big, and they confuse the mapping to the issue tracking system. Producing commits that only create a certain amount of functionality at a time also has a positive effect on a project's documentation aspect. In an ideal situation, every commit has only one link to a feature or an error. Then, it is easy for every code line to backtrack from it into a commit comment, or a possible link to an issue tracking system.

By using these methods, the overall quality of software will be assured. When the quality of the software development process is high enough, the predictions that the predictor give will be more precise. In a sense, the predictor tended to show problems that were already known to the developers and managers. In addition to improving the quality of the software, the data used by the error predictor will be more precise when these methods are used.

7.4.1. Sensitive information

When creating the commit hotness rankings, the prior expectation was that showing names would be disapproved by the developers. The expectation originated from the rationale that showing the name of a developer would cause the developer to fear that the others would blame the developer for creating a "bad" commit.

The actual reception of showing names was surprising: it was embraced by the developers. The developers' rationale for showing the names was that they knew the underlying factors why one would create a "bad" commit. The developers could admit that some of the commits were made more hastily than others. Another reason for a "bad" commit was that some of the developers could have a heavier workload than others at a given time.

8. CONCLUSIONS

This thesis described the current setting of tools and practices in software development, with an emphasis on software testing. A background was given for creating a software error predictor. It included a description of machine learning methods needed for creating an error predictor. This theoretical background was relied on when a software error predictor was made for an embedded system project in Bittium. The steps for creating the error predictor and the output reports were described. An error predictor was successfully made in this project, and it was able to give predictions of the most probable error locations.

The predictions are not useful by themselves. A big effort was made to come up with useful reporting applications for the predicted error probabilities. The most useful and informative report was the error-developer graph, a spin-off visualization made using the extracted data. The graph that combined the error and the author data together did not use the error predictions, although it heavily relied on the algorithmically created error blaming data. When creating the predictor, discussions with developers were had. In these discussions, the errors were often said to come from the lack of interaction between teams, such as insufficient knowledge transfer. The error-developer graph gave insight on these problems.

The error predictor and the reports that were made based on its predictions and the gathered data is the output of this thesis. Furthermore, the data mining part of this thesis caused multiple targets for improvement to come into focus. For example, the different steps made in the data extraction phase brought data quality issues into attention. Some of these issues were acted upon during the project. When inspecting the resulting predictor, the correlation factors gave hints of good and bad practices of software development. These practices were discussed and recommendations were given.

All in all, the pilot project made in Bittium made it clear that the improvements to the software process should be done in the most easily accessible way. The data that was extracted and the final predictions that were made suggested that simple changes should be done in the lowest level of the software process. Improving the quality in the lowest level can offer the best benefit in overall quality of the software. Improvements could be done by encouraging developers to create better constructed commits and to hold on to rigorous code reviewing and unit testing habits. Making these changes improves the error predictions and the overall quality of the software.

One of the original goals was to create a test ranking that is able to prioritize test cases based on their effectiveness of finding an error. Although this goal was not met, the original target of improving the quality of the software was achieved. The process of creating the error predictor gave answers to data-driven decisions concerning software practices. These answers gave proof that a change was needed in the software practices.

9. REFERENCES

[1] Larman C. & Basili V.R. (2003) Iterative and incremental development: A brief history. Computer 36, pp. 47–56.

[2] PricewaterhouseCoopers (accessed 8.7.2014), The third global survey on the current state of project management. URL: `http://www.pwc.com/en_US/us/public-sector/assets/pwc-global-project-management-report-2012.pdf`.

[3] Van Rossum G. (accessed 23.6.2014), rietveld - code review. URL: `https://code.google.com/p/rietveld/`.

[4] Raymond E.S. (accessed 9.7.2014), Understanding version-control systems. URL: `http://www.catb.org/~esr/writings/version-control/version-control.html`.

[5] Beck K. (2003) Test-driven development: by example. Addison-Wesley Professional.

[6] Williams L., Kudrjavets G. & Nagappan N. (2009) On the effectiveness of unit test automation at microsoft. In: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on, IEEE, pp. 81–89.

[7] Catal C. & Diri B. (2009) A systematic review of software fault prediction studies. Expert Systems with Applications 36, pp. 7346 – 7354. URL: `http://www.sciencedirect.com/science/article/pii/S0957417408007215`.

[8] Menzies T., Greenwald J. & Frank A. (2007) Data mining static code attributes to learn defect predictors. Software Engineering, IEEE Transactions on 33, pp. 2 –13.

[9] Turhan B., Menzies T., Bener A. & Di Stefano J. (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14, pp. 540–578. URL: `http://dx.doi.org/10.1007/s10664-008-9103-7`.

[10] Moser R., Pedrycz W. & Succi G. (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on, pp. 181 –190.

[11] Goldstine H.H. & Goldstine A. (1946) The electronic numerical integrator and computer (eniac). Mathematical Tables and Other Aids to Computation , pp. 97–110.

[12] Corbet J., Kroah-Hartman G. & McPherson A. (2012) Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. The Linux Foundation .

[13] Linux Kernel Organization, Inc (accessed 16.11.2015), The linux kernel archives. URL: `https://www.kernel.org/`.

[14] Godfrey M.W. & Tu Q. (2000) Evolution in open source software: A case study. In: Software Maintenance, 2000. Proceedings. International Conference on, IEEE, pp. 131–142.

[15] O'Brien L. (2005) How many lines of code in windows. Knowing. Net URL: `http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/`.

[16] Royce W.W. (1970) Managing the development of large software systems. In: proceedings of IEEE WESCON, vol. 26, Los Angeles, vol. 26.

[17] Westerhoff (accessed 8.7.2014), File:iterative development model v2.jpg. URL: `http://commons.wikimedia.org/wiki/File:Iterative_development_model_V2.jpg`.

[18] Beck K., Beedle M., Van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R. et al. (accessed 5.9.2015), Manifesto for agile software development. URL: `http://agilemanifesto.org/`.

[19] Takeuchi H. & Nonaka I. (1986) The new new product development game. Harvard business review 64, pp. 137–146.

[20] Fowler M. (2006) Continuous integration. ThoughtWorks URL: `http://www.martinfowler.com/articles/continuousIntegration.html`.

[21] Duvall P.M., Matyas S. & Glover A. (2007) Continuous integration: improving software quality and reducing risk. Addison-Wesley Professional.

[22] Humble J. & Farley D. (2010) Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley Professional.

[23] GitHub Inc. (accessed 23.6.2015), Github. URL: `https://github.com/`.

[24] Eclipse Foundation (accessed 8.10.2015), THE OPEN SOURCE DEVELOPER REPORT 2009 Eclipse Community Survey Results. URL: `http://www.eclipse.org/org/press-release/Eclipse_Survey_2009_final.pdf`.

[25] Eclipse Foundation (accessed 8.10.2015), Eclipse Community Survey 2014 Results. URL: `https://dzone.com/articles/eclipse-community-survey-2014`.

[26] Pfleeger C.P. & Pfleeger S.L. (2002) Security in computing. Prentice Hall Professional Technical Reference, 155–156 p.

[27] Cohen J., Brown E., DuRette B. & Teleki S. (2006) Best kept secrets of peer code review. Smart Bear.

[28] Atlassian (accessed 9.7.2014), Bitbucket. URL: `https://bitbucket.org/`.

[29] Meszaros G. (2007) xUnit test patterns: Refactoring test code. Pearson Education.

[30] Ammann P. & Offutt J. (2008) Introduction to software testing. Cambridge University Press.

[31] Cunningham & Cunningham, Inc. (accessed 26.10.2014), Code Coverage Tools. URL: `http://c2.com/cgi/wiki?CodeCoverageTools`.

[32] Dörnenburg E. (accessed 16.11.2015), The softvis collection : Test-to-code ratio chart. URL: `http://www.softviscollection.org/vis/test-code-ratio/`.

[33] SQLite (accessed 16.11.2015), How sqlite is tested. URL: `http://www.sqlite.org/testing.html`.

[34] Utting M. & Legeard B. (2010) Practical model-based testing: a tools approach. Morgan Kaufmann.

[35] Zimmermann T., Nagappan N., Gall H., Giger E. & Murphy B. (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, ACM, New York, NY, USA, pp. 91–100. URL: `http://doi.acm.org/10.1145/1595696.1595713`.

[36] Nagappan N., Murphy B. & Basili V. (2008) The influence of organizational structure on software quality. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on, pp. 521 –530.

[37] Lee T., Nam J., Han D., Kim S. & In H.P. (2011) Micro interaction metrics for defect prediction. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, ACM, New York, NY, USA, pp. 311–321. URL: `http://doi.acm.org/10.1145/2025113.2025156`.

[38] Kan S.H. (2002) Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc.

[39] Hatton L. (2008) The role of empiricism in improving the reliability of future software. Keynote Talk at TAIC PART .

[40] Menzies T., Caglayan B., He Z., Kocaguneli E., Krall J., Peters F. & Turhan B. (2012), The PROMISE Repository of empirical software engineering data. URL: `http://promisedata.googlecode.com`.

[41] McCabe T. (1976) A complexity measure. Software Engineering, IEEE Transactions on SE-2, pp. 308 – 320.

[42] Rogue Wave Software, Inc. (accessed 8.10.2015), Source Code Analysis Tools for Security & Reliability | Klocwork. URL: `http://www.klocwork.com/`.

[43] Synopsys, Inc. (accessed 8.10.2015), Software Testing and Static Analysis Tools | Coverity. URL: `http://www.coverity.com/`.

[44] Watson A.H., McCabe T.J. & Wallace D.R. (1996) Structured testing: A testing methodology using the cyclomatic complexity metric. NIST special Publication 500, pp. 1–114.

[45] Halstead M.H. (1977) Elements of Software Science (Operating and programming systems series). Elsevier Science Inc.

[46] Chidamber S.R. & Kemerer C.F. (1994) A metrics suite for object oriented design. Software Engineering, IEEE Transactions on 20, pp. 476–493.

[47] Nagappan N. & Ball T. (2005) Use of relative code churn measures to predict system defect density. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, IEEE, pp. 284–292.

[48] Śliwerski J., Zimmermann T. & Zeller A. (2005) When do changes induce fixes? SIGSOFT Softw. Eng. Notes 30, pp. 1–5. URL: `http://doi.acm.org/10.1145/1082983.1083147`.

[49] Negara S., Vakilian M., Chen N., Johnson R.E. & Dig D. (2012) Is it dangerous to use version control histories to study source code evolution? In: ECOOP 2012–Object-Oriented Programming, Springer, pp. 79–103.

[50] Williams C. & Spacco J. (2008) Szz revisited: verifying when changes induce fixes. In: Proceedings of the 2008 workshop on Defects in large software systems, DEFECTS '08, ACM, New York, NY, USA, pp. 32–36. URL: `http://doi.acm.org/10.1145/1390817.1390826`.

[51] Wu R., Zhang H., Kim S. & Cheung S.C. (2011) Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, ACM, New York, NY, USA, pp. 15–25. URL: `http://doi.acm.org/10.1145/2025113.2025120`.

[52] Bachmann A., Bird C., Rahman F., Devanbu P. & Bernstein A. (2010) The missing links: bugs and bug-fix commits. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, FSE '10, ACM, New York, NY, USA, pp. 97–106. URL: `http://doi.acm.org/10.1145/1882291.1882308`.

[53] Tian Y., Lawall J. & Lo D. (2012) Identifying linux bug fixing patches. In: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, IEEE Press, Piscataway, NJ, USA, pp. 386–396. URL: `http://dl.acm.org/citation.cfm?id=2337223.2337269`.

[54] Kim S., Zimmermann T., Pan K. & Whitehead E. (2006) Automatic identification of bug-introducing changes. In: Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on, pp. 81 –90.

[55] Bird C., Bachmann A., Aune E., Duffy J., Bernstein A., Filkov V. & Devanbu P. (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, ACM, New York, NY, USA, pp. 121–130. URL: `http://doi.acm.org/10.1145/1595696.1595716`.

[56] Babyak M.A. (2004) What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models. Psychosomatic medicine 66, pp. 411–421.

[57] Akaike H. (1998) Information theory and an extension of the maximum likelihood principle. In: Selected Papers of Hirotugu Akaike, Springer, pp. 199–213.

[58] Hosmer Jr D.W. & Lemeshow S. (2004) Applied logistic regression. John Wiley & Sons.

[59] Mockus A., Zhang P. & Li P.L. (2005) Predictors of customer perceived software quality. In: Proceedings of the 27th international conference on Software engineering, ACM, pp. 225–233.

[60] The Association for Computing Machinery (accessed 15.7.2015), 2003 Gödel Prize — Yoav Freund and Robert Schapire. URL: `http://www.sigact.org/Prizes/Godel/2003.html`.

[61] Freund Y. & Schapire R.E. (1995) A decision-theoretic generalization of on-line learning and an application to boosting. In: Computational learning theory, Springer, pp. 23–37.

[62] Zhou Z.H. (2012) Ensemble methods: foundations and algorithms. CRC Press.

[63] Friedman J.H. (2001) Greedy function approximation: a gradient boosting machine. Ann. Statist. 29, pp. 1189–1232. URL: `http://dx.doi.org/10.1214/aos/1013203451`.

[64] Li C. (accessed 15.7.2015), A Gentle introduction to Gradient Boosting. URL: `http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/slides/gradient_boosting.pdf`.

[65] Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg V., Vanderplas J., Passos A., Cournapeau D., Brucher M., Perrot M. & Duchesnay E. (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, pp. 2825–2830.

[66] Srivastava A. & Thiagarajan J. (2002) Effectively prioritizing tests in development environment. SIGSOFT Softw. Eng. Notes 27, pp. 97–106. URL: `http://doi.acm.org/10.1145/566171.566187`.

[67] Rothermel G., Untch R., Chu C. & Harrold M. (2001) Prioritizing test cases for regression testing. Software Engineering, IEEE Transactions on 27, pp. 929 –948.

[68] Blunck J. (accessed 16.11.2015), Ccm. URL: `http://www.blunck.se/ccm.html`.

[69] Taipale T., Qvist M. & Turhan B. (2013) Constructing defect predictors and communicating the outcomes to practitioners. In: Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on, IEEE, pp. 357–362.

[70] Turhan B. & Kuutti K. (2014) Tracing latent dependencies across software teams through error-handling graphs. CSCW Workshop on GSD: Global Software Development in a CSCW perspective (CSCW'14) .

10. APPENDICES

10.1. SZZ algorithm

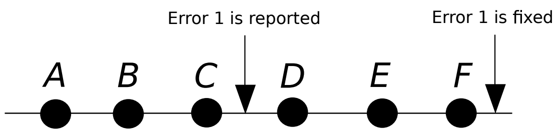


Figure 35: The version history of function multiplyBySix

```
1  int multiplyBySix(int a) {
2      int b = 0;
3      b = a;
4      b = b * 2;
5      return b * 3;
6  }
```

A

```
1  int multiplyBySix(int a) {
2      int b = 0;
3      b = a;
4      b = b * 2;
5      return b * 2;
6  }
```

B

```
1  int multiplyBySix(int a) {
2      int b = 0;
3      b = a + 1;
4      b = b * 2;
5      return b * 2;
6  }
```

C

```
1  int multiplyBySix(int a) {
2      int b = a + 1;
3      b = b * 2;
4      return b * 2;
5  }
```

D

```
1  int multiplyBySix(int a) {
2      int b = a + 1;
3      b = b - 1;
4      b = b * 2;
5      return b * 2;
6  }
```

E

```
1  int multiplyBySix(int a) {
2      int b = a;
3      b = b;
4      b = b * 2;
5      return b * 3;
6  }
```

F

Figure 36: The code history of function multiplyBySix

```
A:    int multiplyBySix(int a) {
D:        int b = a + 1;
E:        b = b - 1;
A:        b = b * 2;
B:        return b * 2;
A:    }
```

Figure 37: The output of `annotate` for version E

To describe the logic of the SZZ algorithm, the steps to determine the fix-inducing change for a known error fix are shown for a trivial C-language function. The version history of the function can be seen in Figure 35 and Figure 36.

It is known that a change f is an error fix for an error with identifier 1 and it transforms version E to version F. The lines that have been changed by f are determined by comparing the file changes. This is done in SZZ by using the SCM to list all the lines that were deleted or added. In SZZ's case, the SCM is CVS and the command used is `diff`. These lines form the change set $L = \{2, 3, 5\}$. Since the change f was an error-fixing commit, these lines are considered to be error-fixing.

Since version E is the latest version without the error fix f, it is used for determining the candidates that have probably caused the error. The CVS command `annotate` is used to determine for every line l of version E the latest version that has touched the line l. Now the versions that have touched the lines in the change set L are candidates to be fix-inducing changes. These are determined to be B, D and E in this example.

The changes that have been committed after the error has been reported are not seen to be fix-inducing. In the example, we can rule out D and E. Therefore, version B is decided to be fix-inducing. In other words, it is said to have caused error 1. The changes ruled out can be partial fixes or candidates for other errors.