Impendulo: A Tool for Analysing Programmer Behaviour

by

Pieter Johannes Godfried Jordaan

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the Faculty of Natural Sciences at Stellenbosch University

Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisors:

Prof. W. Visser Dr. J. Geldenhuys

March 2015

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: January 8, 2015 .....

Abstract

Impendulo: A Tool for Analysing Programmer Behaviour

P. J. G. Jordaan

Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MSc

January 2015

Automated submission systems for Computer Science courses are commonplace today, however, these systems are typically solely focused on grading submissions and their ability to provide analysis and feedback is vastly underutilised.

This thesis seeks to address this by presenting *Impendulo*, a platform for analysing programmer behaviour. Impendulo allows one to study student performance at both a group and an individual level. This is achieved through the use of a customisable set of software analysis tools which Impendulo runs on user submissions. The results produced by the tools are parsed by Impendulo to produce rich feedback in the form of interactive charts, annotated code and detailed reports.

In order to ascertain whether Impendulo's approach is viable, experimental trials were conducted with it throughout its development process. These trials consisted of a group of students using the Impendulo system while solving a set of programming problems. After each trial, the results were studied and used to determine which aspects of the system needed to be improved. At the end of Impendulo's development, all of the experiments were studied again to gain insight into the tested students' programming.

Uittreksel

Impendulo: Analise van Programmeerder Gedrag

("Impendulo: A Tool for Analysing Programmer Behaviour")

P. J. G. Jordaan

Departement Wiskundige Wetenskappe,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MSc

Januarie 2015

Outomatiese inhandigings sisteme vir Rekenaarwetenskap kursusse is vandag 'n algemene verskynsel, maar hierde sisteme is tipies net gefokus daarop om werk te gradeer en hulle kapasiteit om analise en terugvoer te lewer word nie benut nie.

Hierdie tesis beoog om hierdie kwessie aan te pak met die ontwikkeling van *Impendulo*, 'n platform vir die analise van programmeerder gedrag. Impendulo laat gebruikers toe om studente se vordering te monitor op beide 'n individele en 'n groep vlak. Dit word bereik deur 'n student se werk te analiseer met 'n aanpasbare stel analise sagteware. Die resultate wat deur die sagteware geproduseer word, word deur Impendulo ontleed sodat dit terugvoer kan gee in die vorm van interaktiewe grafieke, geannoteerde kode en gedetailleerde verslagte.

Eksperimente is geloop tydens Impendulo se ontwikkeling om te bepaal of ons benadering prakties is. Hierdie eksperimente het bestaan uit 'n groep van studente wat die Impendulo sisteem gebruik terwyl hulle aan 'n stel probleme werk. Na elke eksperiment is die resultate daarvan ontleed en gebruik om te bepaal watter aspekte van die sisteem aandag benodig. Aan die einde van Impendulo se ontwikkeling is al die eksperimente weer na gegaan om insig te kry oor hoe die getoetste studente programmeer.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

- the University of Stellenbosch, where I have studied for the entirety of my tertiary education;
- my study leaders, Willem and Jaco, for their mentorship and support;
- and my parents, Naas and Paulette, for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we describe the need for the system developed by this research. We discuss software bugs, the lack of analysis thereof during a programmer's formative years and the need for feedback in programming courses. The chapter concludes with our approach to addressing these issues and a brief history of the project.

1.1 Background

Software defects Software bugs are an everyday concern in any software development activity and a typical large software project can expect to produce one bug for every 20 lines of code written [1]. These defects eventually account for 50% of the project's cost over the course of its lifespan [2]. When these costs are accumulated across the industry, it is clear that software bugs are hugely detrimental to a nation's economy. In fact, in 2002 RTI International (a non-profit research organisation) estimated that the US sunk $59 billion into eliminating software defects, which is more than 0.5% of their gross domestic product for that year [3] [4].

The prevalence of defects in software projects has led the industry to pour significant resources into detection and prevention [5] [6]. Various methods have emerged to address this issue, including: defensive programming, automated testing, agile development and static analysis. However, investigating software defects during a programmer's formative years is often neglected. Furthermore, the analysis of novice programmers' code has not received much attention as of yet.

Student feedback One way to remedy this, is to expose students to detailed feedback on their coding behaviour. The size and complexity of programming assignments mean that good feedback is very important to computer science students. Unfortunately, these factors also mean that it is difficult for teachers to provide any feedback beyond conventional grading.

When feedback is provided to students, it can be days or weeks after they have completed the assignment. By this time students may have moved on to new work and started working on new assignments. There is therefore little incentive for them to thoroughly study the feedback they have received since it may no longer be directly relevant to the work they are currently focusing on.

Teacher feedback Feedback to teachers in terms of student performance and progress rarely consists of anything beyond the hard data that is the students' grades. This is helpful to an extent, providing a rough overview of how the class as a whole is doing on any given assignment, however, it does have several limiting factors.

Firstly, this data does not present the teacher with a look behind the scenes as to where exactly students are making mistakes. In order to find the causes of these mistakes, the teacher is currently required to examine the results of tests and the student source code in detail.

Secondly, the feedback is usually very one-dimensional in that it only tells the teacher how well students did in terms of a marking rubric. Additional information which can be valuable such as bad style and coding practices are ignored.

Lastly, this information only becomes available after all students have completed their assignments and they have been graded. During the time in which the students are actively working on an assignment, the instructor has very little idea as to how they are actually progressing.

1.2 Proposed Solution

The objective of this research is to design and implement a software analysis toolkit that can help students and teachers to identify and address bad programming practices. We aim to reach this objective by providing a continuous feedback platform which automatically submits, assesses and analyses student work.

The platform uses the results of the assessment and analysis to present both students and teachers with in-depth feedback regarding students' progress. The feedback can take the form of analysis reports, peer reviews and visualisations. After reviewing this information, students and teachers should gain insight into programming issues students may have.

Additionally, our system provides a fine-grained revision history of each student's submissions. Analysis can be conducted on each revision which allows one to pinpoint exactly where a problem emerged during the course of an assignment. Furthermore, it is possible to track the progress of both individual students and groups across any number of assignments.

Our system consists of two distinct components, *Intlola* and *Impendulo*. Intlola is a client-side application which is used to collect data from students and it is named after the Xhosa word for "spy".

While a student writes code in their editor, Intlola records snapshots of their code and sends it to the system's other component, Impendulo. The snapshots are taken whenever the student saves their work and each snapshot consists of the entire saved file as well as associated metadata.

Intlola is completely integrated into the student's editor and is designed to run unobtrusively in the background, requiring minimal user interaction. Indeed, Intlola only requests user input twice: when initialising a session and when ending one.

The name Impendulo also comes from a Xhosa word which translates to "the answer". Impendulo itself consists of the server-side software responsible for receiving snapshots, analysing them and using the results thereof to provide in-depth feedback.

After receiving a snapshot from Intlola, Impendulo stores it in a database and begins processing it. Processing involves running a set of tools, known as the *toolchain*, on the snapshot. The toolchain is designed to be customisable and extensible in the sense that existing tools can easily be configured and new ones added. Tools used by the toolchain are typically compilers, unit testing frameworks and static analysis tools. After each tool is run, its output is parsed and stored in the database.

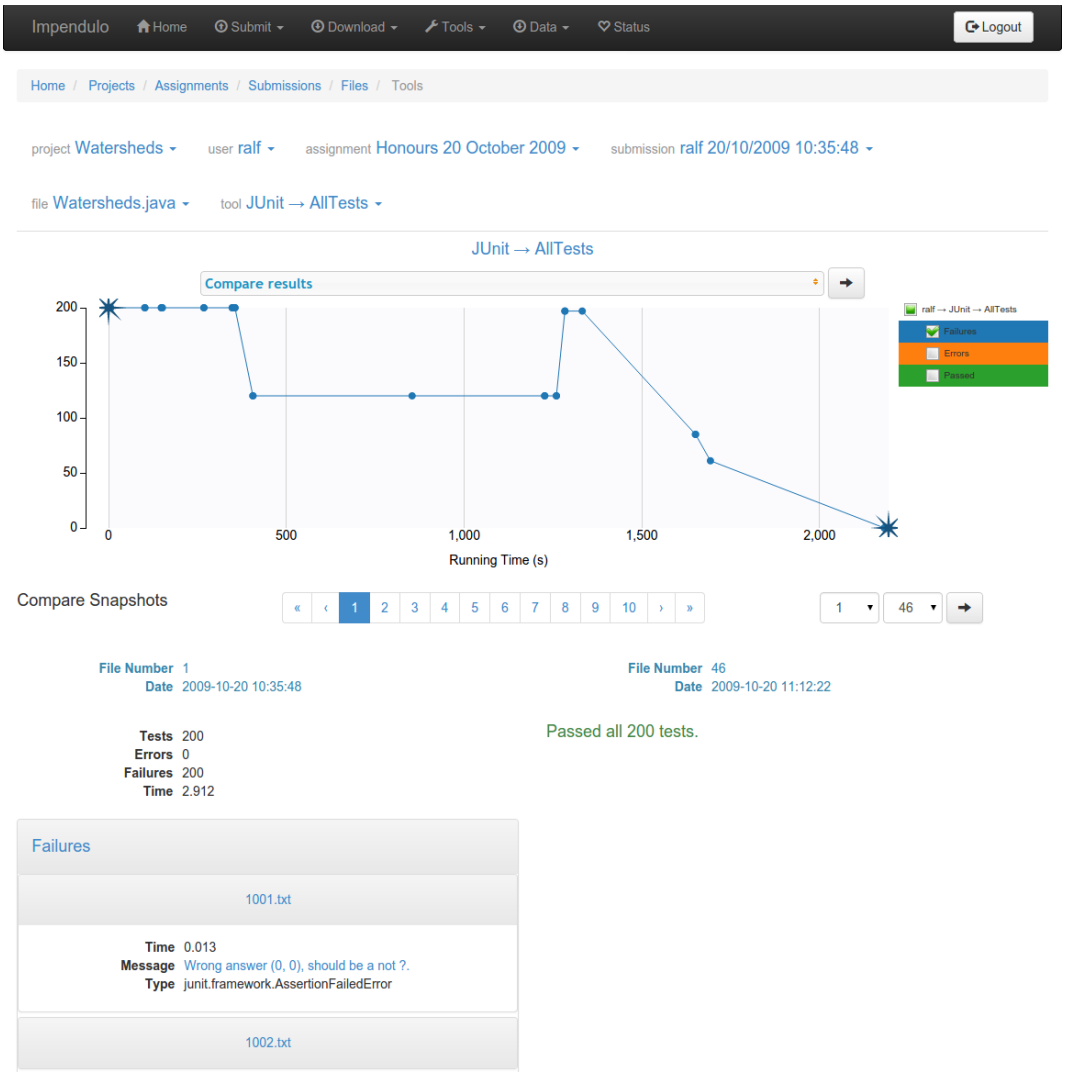
The results are used by our system's web application to provide feedback in the form of tables, charts and reports. These interfaces can each be customised to present users with the data they want. Users are also able to view the results at different granularity levels, for instance by project, submission or individual snapshot. Furthermore, the web application allows users to do detailed comparisons of the data. This includes everything from comparing the results of two static analysis tools to evaluating which group of students performed best on a project.

1.2.1 History

The initial idea behind the project was to collect repositories of programmer's code, test results and static analysis reports. These repositories would then be used to determine where energy should be focused in the development of analysis tools [7]. This would be achieved by looking at the shortcomings of the available tools as well as the information gleaned from the analysed data. Another application of the tool was to be in the teaching of computer science. Here it could be used to:

- Identify and address common mistakes and struggles within a group.
- Provide a good yardstick for measuring group progression during a course.

Figure 1.1: Screenshot of Impendulo's Web Application

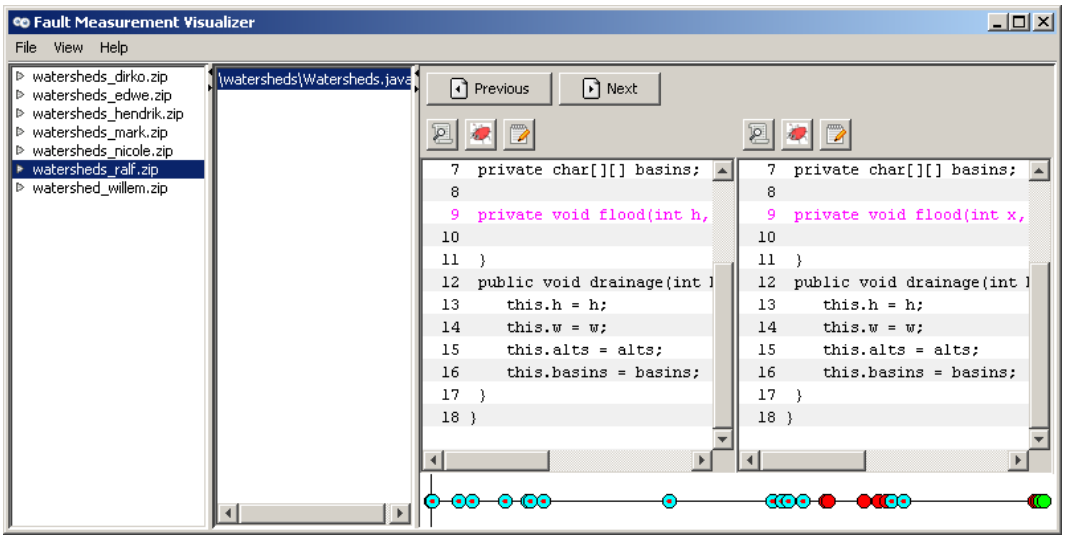


- Provide students and teachers with feedback on their work.

The project was started in October 2009 and a prototype was completed in 2010. The prototype was built for the Java programming language and consisted of an editor plugin and a data visualisation tool. The plugin was able to automatically record and store a student's work whenever it was saved. These recordings were stored as a zip file which could be opened in the visualisation tool as shown in Figure 1.2. This tool, known as the Fault Measurement Visualiser [8], provided the following functionality:

- Charts depicting student progress in terms of compilation success and tests passed.
- Differences between different versions of a student's code.

Figure 1.2: Screenshot of the Fault Measurement Visualiser



- Results of the Findbugs static analysis tool.

A number of experiments were conducted with this prototype and the results led to the development of the current project.

Chapter 2

Related Work

In this chapter we discuss prior work related to this research. For the most part, this consists of automated grading and analysis systems. Due to the fact that there are very many of these systems in existence, this chapter only describes a representative sample. For more in-depth studies see [9] and [10].

2.1 Automated Grading

Automated grading systems have been in use for more than 50 years. They typically consist of an external server to which users submit work which is then graded automatically according to a pre-configured specification.

The first true example dates back to the early 1960s when an automated grading system was in use at the Rensselaer Polytechnic Institute [11]. This system is primarily of historical interest since many of the system's goals and the issues it seeks to address are no longer applicable today.

It is only in the late 1980s when systems emerged which bear a true likeness to Impendulo. The first is an automated grading system developed at the University of Northern Colorado. This software is a Unix-based command-line grading system which can automatically execute programs against test data [12]. The test data is read from standard input and the results of running the tests are written to standard output. The correctness of a program is determined by checking whether it matches a provided set of correct answers. This method of verification means that instructors are required to precisely specify the output format required by an assignment. The evaluation process used by this tool bears similarities to the tool execution process used by Impendulo. Supported languages are: Pascal, FORTRAN 77, C and Modula-2.

Praktomat The next system of interest is Praktomat [13], an automated grading system focused on peer reviewing which emerged in 1998 at the University of Passau and was reimplemented in 2011 at the Karlsruhe Institute of Technology. The idea behind Praktomat is that if students read, review and

assess their peers' work, the quality and style of their own code improves. A basic Praktomat session has the following structure:

1. The student logs into Praktomat using its web interface.
2. They submit their assignment to Praktomat.
3. Their assignment is compiled and tested. Any compilation or test failures result in their assignment being rejected.
4. If the submission is successful, the student has the option of reading and reviewing a peer's work.
5. Hereafter the student may inspect any reviews of their own submission.
6. The student is now free to begin the submission process again.
7. Once an assignment has reached its due date, the instructor runs the final assessment and grade the program.

In order to prevent plagiarism, Praktomat provides students with personalised assignments. This means that the initial time and effort invested in setting up assignments for Praktomat are much higher than when creating assignments for a conventional classroom setup.

All testing in Praktomat is done by means of plugging test data into standard input and then validating the program's output against solution data. Tests can take two guises: public tests and secret tests. Public tests are given to students and they are run every time a student makes a submission to Praktomat. Secret tests are kept hidden from students and are only run during the final assessment.

BOSS The BOSS Online Submission System is a course management tool used at the University of Warwick [14]. Among BOSS's features is the ability to serve as an automated assessment system. It achieves this by:

- Serving as a submission server for student assignments.
- Performing automatic tests for code correctness and quality on new submissions.
- Checking each of these submissions for plagiarism.
- Providing a web interface for marking and issuing feedback.

Student submissions are uploaded to BOSS via a web interface. Students may hand-in multiple submissions, manage them and run tests on these submissions themselves using this interface. Teachers can create, edit and delete

assignments and tests through their own web interface. Furthermore, this allows them to add or remove users from their course, view student submissions and provide feedback on them.

BOSS also has a strong focus on security and privacy. There is a secure sandboxing mechanism in place to ensure that user programs cannot cause any damage to the system. Measures are in place to ensure that data transmission between users and the system is secure.

Quiver The Quiver System is an online teaching and evaluation tool developed at the Appalachian State University in the early 2000s [15]. It allows users to program assignments in C, C++ or Java.

Quiver is designed to be a complete programming environment which builds, maintains and administers programming assignments. However, this means that students are locked into Quiver's closed system. When students are working on a Quiver assignment, they are required to use Quiver's own editor. Furthermore, they are dependent on a remote server to do all compilation and testing. Lastly, assignments done in this environment are required to be completed in a limited time frame.

2.2 Automated Analysis

Automated analysis systems are a more recent development which use built-in software analysis tools to generate detailed feedback about a user's work. Most of these systems also have automated submission and grading capabilities.

ASSYST ASSYST (ASsessment SYSTem) was developed at the University of Liverpool in the mid 1990s [16] as an automated system which can be used to manage all aspects of the grading process. This includes many "housekeeping" operations commonly undertaken when teaching a practical programming course. In a typical course, these tasks may include the following:

- Handling student submissions.
- Managing the files used with the assignments.
- Sending feedback reports back to student.
- Adding grading weights to assignments.

As part of its grading process, ASSYST can evaluate and apply metrics to five areas of assessment:

Correctness A program's correctness is determined through the use of test and solution data. ASSYST runs a program with the provided test data as input. Then it determines correctness by performing pattern

matching of the program's output against the provided solution data. One drawback of this method is that all programs which make use of ASSYST are required to be able to read from standard input and write to standard output. Furthermore, programs are required to exactly match an assignment's input and output specifications.

Efficiency This metric is measured in two ways. Firstly, it measures the CPU time spent executing the user's program. The second method measures the number of statements that the program executes. Both of these are compared to the the model answer's results in order to assign a mark.

Complexity This is simply the program's cyclomatic complexity [17].

Style Grading based on a program's style is done by using an adaptation of style metrics suggested by Berry and Meekings for C programs [18]. The criteria used by this specification includes: program length, use of comments and use of indentation.

Test data adequacy The use of this metric makes it clear that ASSYST has support for test-driven development. ASSYST determines user test quality by measuring the level of statement coverage it reached. Student-provided test data is required to be in the same format as the solution data used to determine program correctness.

Although it is referred to as an automatic system, ASSYST still requires a user to actually initiate the grading of a student or group of students. Once the grading process is complete, ASSYST compiles a report using the aforementioned assessment results. This report is then presented to the user who started the grading process and they have the option of emailing it to the associated student.

Theory Test Assessment Framework The next system of interest is an unnamed assessment framework used at the University of Brighton in an undergraduate Computer Science module [19]. This framework shares a few traits with Impendulo, such as:

- Automated assessment is conducted after a student submits an assignment.
- Feedback is provided to students as soon as this assessment has been completed.
- Integration of several different assessment tools and techniques.
- Results from tools share a common database.
- The ability to handle various different types of assignments.

However this framework does not focus on providing automated assessment of programming assignments, but rather theory tests.

Web-CAT Web-CAT is an automated system developed at Virginia Tech [20]. It is language neutral and has been used with Java, C++, Scheme, Prolog, Standard ML, and Pascal.

A major reason behind its development is to aid the incorporation of software testing as an integral facet of all programming courses at the university. Some of its features are:

- Support for electronic submission of assignments and test cases via a web-based wizard interface.
- Automated grading of student assignments with immediate feedback to students.
- Customisable, scriptable grading actions and feedback generation.
- Emphasis on test driven development. One mechanism for grading students involves using the percentage of instructor tests passed, the percentage of their own tests passed, the code coverage their own tests achieve and the validity of their own tests. Here the validity is a measure of the accuracy of the tests. It determines whether the tests are consistent with the problem statement.
- Provides static analysis by using PMD and Checkstyle to give students additional feedback.
- Code coverage, style warnings and instructor feedback can all be viewed in a single marked-up view of the source code.

Web-CAT has a plugin-style architecture which allows users to extend the base system and add to the available student services. New plugins (known as *subsystems*) can be easily integrated by adding them to Web-CAT's classpath and do not require changes to the source code or recompilation. The subsystem setup is primarily aimed at customising the way in which assignments are submitted and graded.

Marmoset The late 2000s saw the development of Marmoset at the University of Maryland [21]. Marmoset is an automated submission, testing and reviewing system and is designed to be both language neutral and work equally well on both very small and very large projects. The submission system is built as a plugin for the student's IDE and allows them to submit their assignment by simply selecting the appropriate option from the IDE's menu. Marmoset's

plugin is also able to captures code snapshots of students' programming assignments whenever they save their work. This is used for version control and research purposes.

After a student submits their work, it is collected and stored by a *submit server*. This sends it to a separate *build server* which is responsible for compiling and testing the submission. Many of these build servers can be configured to run simultaneously thereby providing greater throughput. One of the key features of Marmoset is the way in which it conducts testing. Each project has a *test setup* which describes how it should be tested. This contains all the extra resources required to compile and run the project's tests. Each test used by Marmoset can be one of four types:

Student tests These are tests which have been written by the submission author.

Public tests These are tests which are run after every submission. Students are given public tests as part of the assignment and have full access to the results of running them.

Release tests These are tests which students must specifically request to be run on their submission. This is due to the fact that students may only request that release tests be run a limited number of times a day. Students do not have access to release tests' code.

Secret tests These are confidential instructor tests which are only run on a student's final submission. Students therefore only receive the results of these tests after the assignment's deadline has passed.

Besides testing, there is also some support for static analysis and code coverage tools. Marmoset is also equipped with a sophisticated code review system.

2.3 Other

Pex4Fun The last system we investigate is Pex4Fun, a system which provides users with a browser-based teaching and learning environment. Pex4Fun uses gamification and interactive techniques to teach students concepts and skills pertaining to programming and software engineering [22].

The main feature used by Pex4Fun to teach programming is known as a *coding duel*. A coding duel is an interactive activity in which students have to attempt to write code which exhibits the same behaviour as a secret implementation. Pex4Fun uses dynamic symbolic execution (DSE) to generate a test suite which covers all feasible paths in both the student's and the secret implementation. The test suites are generated and run whenever the student

saves or submits their work. Creating a coding duel is a straightforward process requiring the teacher to simply write their implementation in a `Puzzle` method. This can then be converted into a coding duel by fulfilling a few basic requirements.

The other type of problem available to students is known as a *puzzle*, which can work in two ways. Firstly, a working version of a program may be provided which the student must study and discern the purpose of. The code can be commented so as to provide students with hints about its purpose. When the student feels they understand how it works, they can click on an `Ask Pex!` button. This executes the code and provides the student with its output. The other type of puzzle provides the student with an incomplete method and a specification for it. The student must then write code so that the method exhibits the desired behaviour.

The exercises which students work on can be chosen in several different ways. Firstly, the student may choose a random problem. Secondly, they may choose to work on a course. A course is composed of several different modules each containing an explanation of the associated concepts as well as some problems. Each module typically contains several puzzles as well as a coding duel. Lastly, they can choose to browse through all coding duels, courses or modules and thereby choose what they want to work on.

Teachers who administrate a course can use it to monitor the progress of students on coding duels. One way in which they are able to accomplish this is to "replay" a student's actions by browsing the historical versions of their implementation. They can then use this to diagnose any issues the students are having with the specific problem. Teachers are also able to view overall class performance and progress. This is provided in the form of a table which contains each student's current result for each coding duel.

2.4 Comparison to Impendulo

In this section we compare the most relevant tools we described in the previous sections to Impendulo.

Praktomat Most of Praktomat's features are not inherently related to Impendulo however there are a few that do overlap. The concept of tests with different privilege levels is one that is also used in Impendulo. Furthermore, both allow users to submit their work multiple times. Lastly, Praktomat's ability to review a peer's code can be likened to Impendulo's commenting feature.

However, the way in which the systems treat multiple submissions is markedly different. Impendulo continuously accepts automatic snapshot submissions while Praktomat requires the student to manually submit their work. Further-

more, Impendulo accepts all submissions while Praktomat rejects submissions if they fail compilation or any tests.

Lastly, in Praktomat reviews are applied to a submission as a whole. This is in contrast to Impendulo where comments are aimed at a specific line or lines in a program.

BOSS Impendulo's Web Application bears marked similarities to BOSS's web interface. They both allow teachers to manage assignments, tests and users. Furthermore, it is possible to view student submissions and provide feedback on them using either interface. Lastly, both provide students with an interface for submitting their assignments.

However, BOSS's web interface does not offer any visualisation or advanced analysis features. Furthermore, Impendulo's web submission interface serves as its secondary submission system. BOSS's web submission system is its only way to submit assignments. Lastly, teacher feedback is much more of a focus in BOSS than it is with Impendulo.

Quiver Impendulo shares a number of features with Quiver.

Firstly, they are both designed to be complete programming environments. Secondly, each uses a remote server to collect, build and evaluate submissions. Thirdly, both are designed so that assignments have a limited timeframe in which they must be completed. Lastly, the systems both have support for multiple programming languages.

However, Quiver locks students into its environment by forcing them to use its editor. This makes students dependent on Quiver's remote server for compilation and testing. Impendulo allows users to program in any editor which has the Intlola plugin. Impendulo's assignments also often provide students with tests.

ASSYST In a similar manner to Impendulo, ASSYST attempts to manage all aspects of the grading process. Furthermore, it is capable of generating reports using different metrics. This can be viewed as a simpler version of Impendulo's method of generating a report for every tool in its toolchain.

However, ASSYST is not truly an automated system unlike Impendulo, since it requires direct input from a user to initiate the assessment process. Furthermore, the metrics it uses for its reports are fixed whereas Impendulo comes with a configurable and extensible toolchain.

Web-CAT Like Impendulo, Web-CAT is a feedback focused automated grading system. Furthermore, it also has support for analysis tools and can provide the user with an annotated view of their code. However, it is not a simple task to expand the available set of analysis tools and the presentation of their results is not as advanced as Impendulo's.

Web-CAT is much more focused on grading when compared to Impendulo. It allows the user to fully customise how assignments are setup and evaluated, while Impendulo is more focused on it having an extensible toolchain.

Marmoset Similarly to Impendulo, Marmoset is a language-neutral automated submission, testing and reviewing system. Furthermore, its submit server-build server setup is very similar to Impendulo's Receiver-Producer system.

Marmoset also utilises an IDE plugin for submission purposes. Furthermore, the plugin does also have snapshot capturing capabilities. However, unlike Impendulo, Marmoset does not use the snapshots as part of its submission system.

A key feature of Impendulo is its support for external tools. Marmoset does have some support for static analysis and code coverage however, this come from tools hardwired into the system.

Marmoset is more oriented towards being a grading system. This means that an important aspect of Marmoset is its use of different types of tests. Impendulo also supports this to a degree. However, Marmoset's setup is more advanced and ingrained into the system.

Pex4Fun It may seem that Pex4Fun does not bear too many direct similarities to Impendulo, but under the hood there are a number of related concepts.

Firstly, Pex4Fun has the ability to store a user's coding history, just like snapshots in Impendulo. This allows teachers to browse through all of a student's attempts at solving a problem. This is very similar to browsing through individual snapshots in Impendulo. However, in Pex4Fun the snapshots are taken whenever the user tests their solution while Impendulo makes a recording after every user save. Impendulo's recording method should therefore capture a larger number of snapshots. Another difference is that Pex4Fun only gives you the student's code when browsing their coding history, while Impendulo provides various other tool results for every snapshot.

Secondly, teachers are also able to view the current status of their class with regards to their attempts at solving the course's problems. In Impendulo you are able to view the current status of all the students working on an assignment.

Lastly, the puzzle exercises are something which can be used in Impendulo, and the second type of puzzles often are. However, in Pex4Fun problems often do not have a clear specification and it is part of the challenge to decipher it from an existing implementation. This is in contrast to Impendulo where clearly defined problems are the only way in which assignments have been set up thus far.

Chapter 3

Design

In this chapter we describe the design of the software analysis toolkit. We first look at how the system functions as a whole, before describing the design decisions behind individual components in greater detail.

3.1 Overview

In Figure 3.1 an overview of the system's components and how they function is shown.

The first point of interest is Intlola, the editor plugin which submits the student's work to Impendulo. When the student starts working, Intlola opens a connection to a remote server and starts a new session for the student. This server is known as the *Submission Server*. While the student programs, Intlola takes a snapshot of their code every time they save it and sends these snapshots over the established connection to Impendulo.

For each new connection the Submission Server receives, it creates a new *Submission Handler*. Collectively, the Submission Server and its Submission Handlers are known as the *Receiver*.

The Submission Handler uses the connection to communicate with and receive snapshots from Intlola. After being created, a Submission Handler adds a new submission job to the *Submission Queue*. A submission job simply specifies that the submission must be processed. The Submission Queue holds submission jobs until they can be processed. Adding a new submission job also spawns a new *File Queue* for that submission. This File Queue is used to hold jobs for every snapshot in the submission.

Whenever a Submission Handler receives a snapshot, it stores the snapshot in Impendulo's database and creates a file job for the snapshot which is added to the submission's File Queue. The file job simply specifies that the snapshot must be processed by running a set of configured tools on it.

The *Processing Server* is the server responsible for handling submission jobs. Any number of Processing Servers can be active at any given time and

each one manages a limited number of *Submission Workers*. A Submission Worker is the entity actually responsible for executing individual submission jobs. Whenever a Processing Server has a Submission Worker slot open, it retrieves a submission job from the Submission Queue and spawns a new Submission Worker for it.

After being started, the Submission Worker can retrieve its submission's first file job. The Submission Worker is then required to load the job's corresponding snapshot from the database. This snapshot is run through the Submission Worker's toolchain. The toolchain is composed of all the tools which are configured to be used with the submission's assignment. After each tool is run on a snapshot, the tool's result is stored in the database. The next file job is retrieved when all of the tools have run on the snapshot. Once all of the snapshots in the job are processed, the Submission Worker is terminated and the submission's File Queue deleted. This opens a Submission Worker slot for the Processing Server and it can start another submission job.

A tool in Impendulo is any piece of software that implements Impendulo's tool interfaces.

The first interface governs how the tool is used which requires the tool to specify how it must be run from the command-line. Furthermore, the tool must also specify how it can be configured and it must parse its output into a format which can be used by Impendulo.

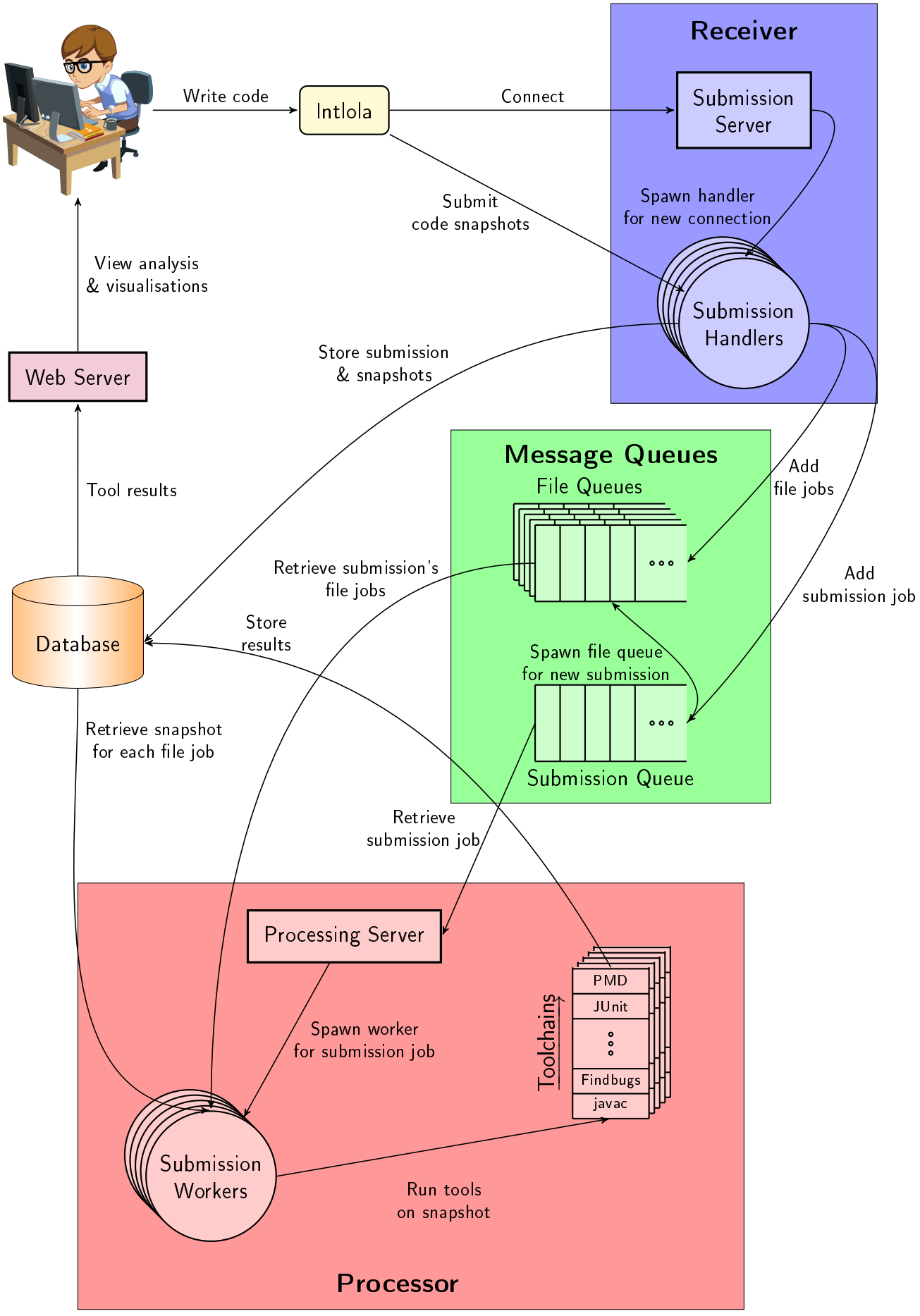
The second interface is used to specify how its results should be rendered to users. The tool must provide an HTML segment, typically a `div` element, which is used to display the tool's results in detail. Optionally, the tool's results can provide a number of numerical values which are used by Impendulo to provide feedback in the form of charts and tables.

The *Web Application* is where the tool's results are presented to users. Users are able to view the results at different levels. At the lowest level, they view a tool result for an individual snapshot, while at higher levels they can view the results aggregated by submission, assignment or project in the form of tables and charts.

3.2 Features

Automation Impendulo is designed to be a fully automated system. This means that students' work is automatically recorded and submitted to an external analysis server as they are working. The only input required from the students is for them to initiate their session by logging in and terminate it when they are done. Impendulo also runs analysis and grading tools automatically on snapshots as soon as they are received. The results of these tools are automatically converted to feedback reports and visualisations by Impendulo.

Figure 3.1: Overview of Impendulo's Architecture



Customisation Extensibility and customisation is very important to Impendulo. The way in which tools are setup in Impendulo makes it a straightforward task to add new tools. Each assignment can also be configured to run tools in a specific manner. Impendulo is designed to be language neutral. Tools and projects can therefore be aimed at any language.

Feedback The Web Application provides a rich feedback interface to both students and teachers. It allows users to view tool results at different granularities. This ranges from inspecting individual results to looking at an aggregation of all results for a specific project. Furthermore, results can be visualised in the form of interactive charts. These charts also allow the results of different tools to be compared by plotting them against each other.

Administration The Web Application provides a powerful administration interface for managing assignments, users, tools, and submissions. The primary function of the administration interface is to allow teachers to create and configure new projects and assignments. Additionally, it allows administrators to edit and delete data.

Robustness The system is designed to continue functioning in almost all circumstances. The architecture ensures that problems such as errant student code or component failures does not result in the system as a whole losing functionality.

Modularity The components that comprise Impendulo are designed to be completely independent of each other. This ensures that the number of active components of any component type can easily be altered as is required.

Scalability Impendulo has the ability to easily scale its processing power up or down. This is due to its modular design which makes it trivial to add or remove new components across multiple nodes.

Portability The framework does not rely on any operating system specific functionality. Therefore, it can be installed on most commonly used systems. Furthermore, its data can be easily transferred across different installations.

3.3 Intlola

Intlola is currently only available for the Eclipse IDE [23]. However, it is designed so that its functionality can easily be ported to other IDEs and text editors. This was achieved by creating a clearly defined communication protocol over which it can communicate with Impendulo. Therefore, an Intlola

plugin for another editor can easily be created by simply using this protocol. For a more in-depth discussion about the protocol see subsection 4.2.2 and Appendix D.

Intlola's current implementation does not use any operating system specific functionality, which allows it to be used on any operating system supported by Eclipse. Intlola is language neutral and has thus far been successfully tested with Java, C, C++ and Python.

Whenever a save is made during an Intlola session, Intlola takes a snapshot of the entire saved file as well as the time it was saved at, its name and its package. The snapshots which Impendulo takes, fall into three categories.

Firstly, they can be the main solution files of an assignment. These files are tested by Impendulo and should therefore implement all the functionality specified by the assignment.

Secondly, a snapshot can be a helper file used by the main file to achieve its desired functionality. These files are normally created by the student and, although they are not tested directly by Impendulo, they are still analysed.

Lastly, in some assignments, students are provided with empty test files in which they can write tests for their solution. Changes to these user test files are also recorded and sent to Impendulo for processing. They are typically used to determine how well students tested their code by looking at the code coverage they achieve.

Intlola's approach to recording code does come with a slight drawback. The intervals and regularity at which saves are made can vary widely from student to student. Furthermore, this rate is not indicative of how well a student is progressing on an assignment. This means that we end up with submissions with widely disparate snapshot counts, which makes it more difficult to do comparisons between them.

An alternative is to rather capture snapshots at regular time intervals. However, this approach comes with its own complications. Firstly, Intlola may send snapshots when no changes have been made to the code. Secondly, the snapshots sent are less likely to compile than when using student saves. This is because students normally save their work when it does not have any compilation errors. Lastly this approach can miss important changes in the student's code since there is no way to know at what time intervals the student will make major changes to their code.

Intlola is designed to be straightforward to install and use. The installation process achieves this by simply using Eclipse's software installation wizard. Intlola itself addresses the issue by keeping user interaction to a minimum. Whenever user input is required, it is obtained by using simple and well-defined dialog boxes.

By minimising user interaction, Intlola also ensures that the student is not distracted from their assignment. All interaction with the student is conducted before they have started their assignment or once they have completed it, which further establishes Intlola's unobtrusiveness.

Intlola is designed so that it can be used with different network settings or *modes*.

File Mode When Intlola uses *File Mode*, it submits snapshots to Impendulo as soon as they are recorded. This mode allows feedback to be generated while the student is still busy working on their submission. However, File Mode requires Intlola to maintain an open network connection to Impendulo throughout the session.

Archive Mode In *Archive Mode*, all snapshots are stored locally while the student is working. Intlola compresses these snapshots into an archive file and submits the archive after the student finishes their assignment. Therefore, Archive Mode only requires Intlola to connect to Impendulo at the end of the session when this file is sent. Furthermore, less data is transferred due to the archive's compression.

Offline Mode As the name suggests, *Offline Mode* allows the student to use Intlola completely offline. In this mode, Intlola saves all snapshots in an archive which can then be manually uploaded to Impendulo at a later stage. Intlola never communicates with Impendulo in this mode and the upload is done using Impendulo's web interface.

3.3.1 Usage

In order to start using Intlola, the student needs to create a project in which they can work on their assignment. Typically this is done by importing a project skeleton which contains the base files and configurations needed to complete the assignment. The most important files found in the skeleton are:

Solution file This is the file in which the assignment must be coded and it is found in every skeleton.

Test files These files provide testcases which the student can use to determine their implementation's correctness. They may be accompanied by data files which contain the actual testcases. Not all assignments provide the student with test files.

User test file A file in which the student may write testcases for their implementation. This file is not always found in the skeleton.

An Intlola session is started by right clicking on the project in the *Project Explorer* or the *Package Explorer* view and selecting `Intlola` → `Record`. This starts Intlola's wizard which allows the student to specify various settings which govern Intlola's execution. In the first step the student is presented with a dialog from which they can choose which mode to run Intlola in.

Figure 3.2: Logging in with Intlola

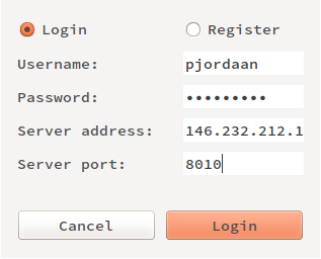
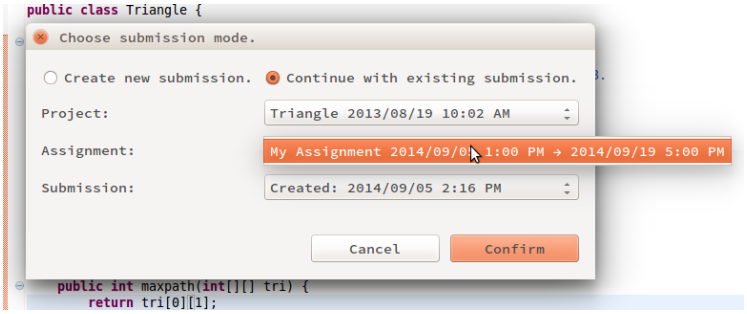


Figure 3.3: Continuing with a Submission



File Mode After choosing File Mode, the student is required to complete what is known as the **Configuration Process**. The Configuration Process is composed of two steps:

1. Firstly, the student is prompted to login to or register with an Impendulo server of their choice (Figure 3.2). They are required to provide the server's address and port, as well as a username and password.

2. Secondly, they are required to choose the type of submission they wish to make. They can either continue from an existing submission (Figure 3.3) or they can choose to create a new submission. For the first choice they are provided with a list of submissions which they have previously worked on and whose assignments are still active. For the second option, they are provided with a list of active assignments to choose from.

Once this process is complete, coding starts and Intlola records their work. When the student is happy with their solution to the assignment, they must end their session. This is done by right clicking on the project once more and selecting `Intlola` → `Record Stop`.

Archive Mode Upon selecting Archive Mode, a new session is started immediately. Once the student is finished with the assignment, they can end their session as detailed in File Mode. This triggers the aforementioned Configuration Process and after the student completes it, the session is truly finished.

Local Mode This mode functions in the same manner as Archive Mode until the student ends their session. At this point a dialog prompts the student to choose where they want to save their snapshots. The snapshots are saved as a zip file which the student can manually upload to Impendulo via its Web Application at a later time.

3.4 Impendulo

While individually small, collected snapshots of a class of students represents a large amount of raw data. This data needs to be collected, processed and presented in an efficient manner, which is where Impendulo comes into play.

Receiver The Receiver is designed to be a robust and stable interface between Intlola and Impendulo. This is achieved by completely separating the component responsible for accepting connections (Submission Server) from the one that interacts with the connections (Submission Handler). After being launched, each Submission Handler runs completely independently inside of their own thread. Therefore, if any Submission Handler should fail, it cannot impact on the execution of the Submission Server.

The execution of a Submission Handler follows these steps:

1. Open a new session.
2. Authenticate the student's credentials.
3. Gather preliminary metadata from the student.
4. Add a new submission job to Submission Queue. This also creates a new File Queue for the submission.
5. Continuously receive code snapshots from the student as they work.
6. Store each new snapshot in Impendulo's database.
7. Add a file job for each snapshot to the submission's File Queue.
8. Once the student decides to end their session, gracefully close the connection.
9. Add a special job to the File Queue to indicate that the session is finished.
10. Terminate the Submission Handler.

Processor Even more so than the Receiver, the Processor is designed to continue functioning even when it encounters failure.

Firstly, in a Processor component, each Submission Worker runs independently of the Processing Server. Therefore, if a Submission Worker fails, the Processing Server continues to function normally. The only input the Submission Worker receives from the Processing Server is the identity of the submission that it must process.

Secondly, each Submission Worker is completely independent of every other worker. They cannot interfere with each other in any manner. This is achieved by making every Submission Worker responsible for their own submission and therefore they have their own set of snapshots. This is further enforced by creating a File Queue for each submission. The File Queue is the only way a Submission Worker obtains the identity of the snapshots it must process and each File Queue is only accessible by a single Submission Worker. Therefore, only one Submission Worker can process a snapshot at any given time. Furthermore, each Submission Worker is provided with their own toolchain and a unique directory in which they are allowed to run their tools. This ensures that the execution of two different Submission Workers' tools cannot interfere with each other.

Thirdly, the Submission Worker is designed so that the effect that the execution of each tool in the toolchain can have on other tools is minimal. Specifically, any failures experienced after running one tool does not have any bearing on the use of any other tool. Whenever a failure is experienced, it is simply reported and recorded. Furthermore, tools are not able to hinder a Submission Worker's progress greatly by running for too long and creating a bottleneck. Impendulo prevents this by specifying that each tool must finish executing within a predefined time limit.

Lastly, it is possible to run multiple Processors simultaneously. Therefore, even if one Processor should fail, Impendulo can continue functioning. This is possible because each Processor functions independently of the rest of Impendulo. They cannot interact with other Processors and can only interact with other components within Impendulo over the message queue. This ensures that the other components are completely unaware as to how many active Processors there currently are.

Being able to easily run multiple Processors simultaneously allows Impendulo's Processor component to be very scalable. Each Processor only needs to know the IP address of Impendulo's message queue and database in order to be launched. It is therefore a trivial process to add new processors to Impendulo as required. These processors can be located anywhere as long as they have network access to Impendulo's message queue and database.

The Submission Worker's toolchain is designed to be extensible and flexible. As described in section 3.1, the way in which tools are integrated into Impendulo means that Impendulo's set of available tools is itself customisable. Furthermore, this tool setup also allows tools to specify a configuration step

which is used to customise the tool's execution for specific assignments. Lastly, the exact tools present in the toolchain can be configured on a per-assignment basis.

Web Application As discussed previously, the most important function of the Web Application component is to provide an interface for visualising and interpreting data produced by tools. In order to make this interface powerful as possible, flexibility, customisation and interactivity are prioritised.

Firstly, we address flexibility by allowing users to view data at different granularities. Furthermore, at each level the user is able to choose between different representations of the data such as tables, charts or reports. This should ensure that the user can view the data in a format which allows them to extract the information they need.

Secondly, tables and charts are interactive and customisable. Each table allows the user to choose exactly what should be displayed in it. The available values which the user can choose from depends on the level at which the table is shown. For the assignments level this would include information about each assignment (name, date, etc.) and the aggregated tool results for all submissions in that assignment. Furthermore, various additional measures may be available such as:

- Average snapshot line count for the assignment's submissions.
- Average submission session duration.
- Average number of submission snapshots, program executions and user testcases.

Charts are able to visualise tool results as well as the additional measures mentioned above. These metrics can then easily be compared to each other due to the interactive and customisable nature of the charts.

The data generated by the tools is also available in a textual format should a more detailed analyses be required. The way in which this is presented depends on the tool, since their HTML file is used to display the results.

As an additional view, we provide the user with the source code of each snapshot. In order to make this more meaningful, a powerful code annotation mechanism is integrated into this view. Annotation is done by attaching textual information to specific lines in the code. Annotations can be comments provided by teachers or peers as well as analysis information provided by the various tools which have processed this file.

The next important function which our web application fulfills is the administration of data. Firstly the Web Application facilitates the creation of new projects and allows easy configuration of tools specific to the project. In addition to this, it is possible to create multiple tests and project skeletons for each project. Furthermore, assignments can be configured so that they are

only available for a specific timespan. This allows Impendulo to be used for assignments that have deadlines.

The Web Application also provides functionality for submitting archive files containing snapshots recorded with Intlola. These are archives created by Intlola when it is run in offline mode. The Web Application sends these snapshots to a Processor for processing.

Message Queues Impendulo uses *message queues* for inter-component communication. A message queue is a software component which facilitates inter-process communication by providing an asynchronous communication protocol. This means that it can hold communication data from a sender until the receiver is ready to use the message. In Impendulo we use several different message queues, each with a unique name and all sharing a common host.

The most important message queues are the Submission Queue and the File Queues. The Submission Queue holds jobs for all submissions which still have snapshots which need to be processed, while the File Queues are temporary queues which hold jobs for a specific submission's unprocessed snapshots. Using File Queues ensures that snapshots can be processed as soon as they are received. Furthermore, it allows snapshots to be processed in the order in which they are received.

By using message queues, we ensure that Impendulo's components are loosely coupled. This is because each component only needs to be aware of the location of the message queues and the database in order to function.

The message queue protocol used by Impendulo is known as Advanced Message Queuing Protocol (AMQP). AMQP specifies a programmable network protocol which allows client applications to communicate with each other via middleware brokers. AMQP provides a number of message delivery guarantees as well as authentication and encryption [24].

RabbitMQ is used as the message broker middleware in Impendulo's AMQP setup. This means that it implements the AMQP standard, providing a multilanguage, cross-platform messaging interface. This interface serves as a common platform which clients can use for sending and receiving messages as well as providing a safe place for messages to live until they can be received [25].

Communication over the RabbitMQ interface is conducted in a Producer-Consumer manner. A Producer creates a piece of work which is added to a queue. Later, an idle Consumer asks for a new task. The task is dequeued and sent to the Consumer which begins working on it.

Database Impendulo uses MongoDB [26], a NoSQL, document-oriented database, to store its data. Its internal structure consists of documents with dynamic schemas stored within entities known as collections. The documents' structure closely resemble that of a JavaScript Object Notation (JSON) object.

A collection can be viewed as a relational database's table and a MongoDB document as one of the table's rows.

The schema-less nature of MongoDB allows us to store data with differing structures in the same collection. This is well suited to storing the results of different tools since Impendulo does not define a specific structure for results. Furthermore, this enables us to easily make changes to the structure of data stored in the database.

The way in which Impendulo functions means that it tends to exert a high write load on its database. This is due to a lot of snapshots being stored concurrently by the Receiver and even more tool results being saved by Submission Workers. MongoDB is able to manage this load since it offers very good performance in high write load situations.

Chapter 4

Implementation

This chapter describes the implementation of the various components which comprise Intlola and Impendulo's architecture.

4.1 Intlola

The Eclipse version of Intlola is built using the Eclipse Plug-in Development Environment (PDE), a framework for developing Eclipse plug-ins and related software. Intlola itself consists of three primary components, the Plugin, the Controller and the Processor.

Plugin The Plugin is used to keep track of any changes made by the user to their project, by interacting with the Eclipse PDE API. Whenever it detects a change, the Plugin inspects the modified resource and determines whether the change should be recorded and sent to Impendulo.

Controller The Controller specifies how interaction with the user takes place and how information is retrieved from them. The Controller then uses this information to configure how the Processor operates.

The first interaction the user has with the Controller is when they choose which mode to run Intlola in. Secondly, if not running in offline mode, it handles the user login or registration process. Lastly, the Controller allows the user to choose which assignment to work on. If they have an existing submission in this assignment they can choose to continue with it, otherwise a new submission is created for them.

Processor The Processor component is responsible for conducting all communication with Impendulo. The main purpose of this communication is to send snapshots to Impendulo, but it also creates and stores snapshots and builds archives.

The Processor uses threads to perform its tasks, but each task must be performed sequentially. Therefore, the Processor uses a single worker thread to complete its tasks. In order to ensure that all tasks are completed, the Processor uses `java.util.concurrent.ExecutorService` interface which provides functionality for managing asynchronous task progress and termination [27].

4.2 Impendulo

The backend infrastructure of Impendulo is primarily written in Go [28]. However there are segments of it that are built with Java and bash. The web interface is written in JavaScript, HTML and CSS.

4.2.1 Nomenclature

goroutine A `goroutine` is a term used in Go to describe functions which are run concurrently with other functions. Typically a group of `goroutines` is multiplexed onto a set of threads. Whenever one `goroutine` blocks a thread, the run-time automatically moves the remaining `goroutines` on that thread to another thread. They can be thought of as very lightweight threads [29].

channel Go makes use of the message-passing communication model for its concurrent programming. In Go, a `channel` is used to implement this model. `Channels` allow for communication and data exchange to take place between `goroutines`. Go allows multiple goroutines to read from and write to the same `channel`. `Channels` are synchronous by default, but can be made asynchronous by allocating a buffer size to them. This buffer allows the channel to function as a queue until the buffer's capacity is reached. When the capacity is reached, the `channel` blocks until an item is read from the `channel`.

4.2.2 Receiver

As detailed in chapter 3, the Receiver consists of two components: The Submission Server and the Submission Handler.

Submission Server The Submission Server is implemented as a simple TCP server which continuously listens on a preconfigured port for new connections from Intlola. For each new connection it receives, it spawns a new SubmissionHandler and launches it as an independent `goroutine`.

Submission Handler As specified in section 3.4, Submission Handlers are responsible for managing an entire Intlola session. The communication protocol used for the interaction between a Submission Handler and Intlola is implemented as a simple Application Programming Interface (API). The API's commands are called by sending a JSON object to the Submission Handler with the necessary arguments. Each JSON object contains the name of the command in the object's "request" field. Listing 4.1 shows an example of a call to the API's `login` function.

Listing 4.1: Example API call

```
{''request'': ''login'',
  ''user'': ''jonny57'',
  ''password'': ''totally_impregnable_defense'',
  ''mode'': ''file_remote''
}
```

The only occasion on which Intlola sends non-JSON data is when it transfers snapshots to the Submission Handler. On this occasion, after sending the snapshot's metadata in a JSON object, the actual snapshot is sent as a stream of raw bytes.

The fact that the data transmitted here is unencrypted is an immediate cause for concern and sending passwords in plain text is especially worrying. However, ensuring secure client-server communication is not a goal of this research and therefore adding the additional security measures is beyond the scope of this work.

For a more detailed description of the API, see Appendix D.

4.2.3 Processor

Processing Server The Processing Server controls the operation of a single Processor and manages how submissions are processed by Submission Workers. Whenever it has an open worker slot, it attempts to retrieve a submission job. If successful, it launches a new Submission Worker in its own `goroutine`. The Processing Server has a single buffered `channel` which it uses to detect when a Submission Worker has finished processing its submission. Each Submission Worker has access to this channel and uses it to signal their completion.

Submission Worker A Submission Worker is the core component at the heart of all of the snapshot processing done by the Processor. It retrieves new file jobs from its submission's File Queue and processes them. This continues until it receives a special job indicating that the submission is completely processed. File jobs can be for normal snapshots, archives of snapshots or user test snapshots.

If the job is for a normal snapshot, the Submission Worker processes it in the following manner:

1. Retrieve the snapshot from the database.
2. Save the snapshot in the Submission Worker's allocated directory.
3. If the Submission Worker has a compiler in its toolchain, use it to compile the snapshot.
4. Run the Submission Worker's tools on the snapshot. For each tool, the following steps are carried out:
   a) Check if the snapshot already has a result for this tool. If it does, skip this tool. Otherwise continue.
   b) Run the tool on the snapshot.
   c) Save the result of running the tool. This result can either be the actual result of running the tool or it can be an error. The error can either be directly produced by the tool or it can be a timeout error due to the tool taking too long to finish executing.
   d) Move on to executing the next tool in the pipeline.
5. This snapshot has now been processed and the next one can be requested.

When processing an archive of snapshots, the Submission Worker firstly extracts all of the snapshots. Next, each snapshot is stored in the database. These snapshots are then processed according to their type (normal or user test). Lastly, the archive is deleted from the database.

User tests are processed as if they are JUnit test file used by the JUnit tool. This means that when the processor receives a user test file, it runs the JUnit tool, with the user test as the configured test file, on each normal snapshot in the submission. In the same manner, code coverage is also run for the user test file on each normal snapshot.

Toolchain We have thus far only created a fully functioning toolchain for the Java programming language. The toolchain currently consists of:

javac This is the Java compiler and is used to compile a snapshot's source code into Java bytecode. It is the first tool run by Impendulo [30].

JUnit This is a framework for writing repeatable tests for the Java programming language [31]. We use JUnit to determine the correctness of submissions and to allow users to write their own tests. Typically an Impendulo assignment which uses JUnit has two different test files, `EasyTests.java` and `AllTests.java`. `EasyTests.java` contains a few basic testcases which should not be too difficult for the user to pass and helps them to

see whether they are on the right track. `AllTests.java` contains a large number of more difficult tests which are used to determine how well the user did on an assignment. A user's own tests are normally written in a `UserTests.java` file which is provided in the assignment's skeleton.

Jacoco The measurement of the code coverage achieved by tests is conducted by Jacoco [32]. The code coverage obtained by a test describes how much of a program is tested by it. Jacoco supports instruction, line, branch, method, class and cyclomatic complexity [17] coverage counters.

Findbugs The first static analysis tool in our setup is Findbugs which is used to identify potential errors in Java byte code [33]. Errors detected by Findbugs are classified according to severity and the types of errors checked for can be configured via rule sets.

Checkstyle Our next static analysis tool is Checkstyle, which is used to determine whether source code adheres to coding standards [34]. The coding standards used by Checkstyle can be customised to fit most code conventions.

PMD The last static analysis tool is PMD and it supports multiple languages including Java and JavaScript [35]. PMD's analysis is rule-set based allowing it to be easily configured to detect potential bugs, bad style and duplicate code.

JPF Java Pathfinder (JPF) is a model checker for Java programs and it is used to find defects in programs by systematically exploring all execution paths through the program [36].

lc Lastly, we have `lc` which simply uses the unix `sed` [37] and `wc` [38] utilities to count the number of non-empty, uncommented lines in a snapshot.

Additionally there is one utility which is not directly integrated into the toolchain, but which still produces data for the web application from snapshots. This is the unix `diff` tool which calculates the differences between two snapshots. This is calculated on-the-fly because it is possible to display the `diff` result between any two snapshots in a submission. This makes it unfeasible to calculate all these results beforehand.

For a guide to adding new tools to the toolchain see Appendix A.

Monitor The Monitor keeps track of the current status of all of Impendulo's Processors by maintaining a single structure which combines the status of each Processor into one. This structure consists of a list of all the submissions which are currently being processed. Each of these submissions also maintains lists of their snapshots which are currently being processed or which still need to be processed.

The Monitor's status is updated by Processors via one of Impendulo's message queues. Therefore, the Monitor must continuously listen for update requests and alter its status accordingly. Each type of request has a specific action associated with it which the Monitor carries out upon receiving it. Firstly, there are requests which signal that a submission's processing has started or stopped. This results in the submission being added to or removed from the list of submissions being monitored. The other type of request modifies a submission's list of incomplete files and is used when a snapshot arrives or is finished processing.

The Monitor is actually an independent component from the Processor, however, since it is used to convey information about the Processor to other components in Impendulo, we consider it to be part of the Processor.

Multiple instances of the Monitor can run simultaneously which ensures that even if one should fail, Impendulo can still keep track of its processors.

4.2.4 Web Application

The Web Application is the component of Impendulo with which users interact with the most. It provides the user with a platform for visualising, analysing and interpreting data, known from here on as the Analysis Interface. Furthermore it has many administrative functions which are accessible from what is known as the Administration Interface.

4.2.4.1 Analysis Interface

The Analysis Interface is composed of a hierarchical structure which allows the user to browse Impendulo's data at different granularities. In Table 4.1 we can see that the hierarchy consists of six different levels with two ways to view it. At the top level, the user can choose to browse either by project or by student. From here they can drill-down into the data by selecting a specific project or student. This takes the user to the assignments level, which, depending on the selection, either consists of all of the assignments associated with the selected project or all of the assignments the selected student has participated in. From this level the user can select an assignment which allows them to view all the submissions made for it. However, if the user is browsing by student, only the student's submissions for this assignment are shown. The next level available to the user, the files level, can be viewed by selecting a specific submission. The files level does not consist of the submission's individual snapshots, but rather the different files with regards to their name. Each of these files is therefore actually a group of snapshots which share the same name. Selecting one of these files takes the user to the lowest level at which each of the individual snapshots that share the file's name can be viewed.

Level	Project	Student
1	Projects	Students
2	Project Assignments	Student Assignments
3	Assignment Submissions	
4	Submission Files	
5	File Snapshots	

Table 4.1: The Analysis Interface Hierarchy

Each level is therefore the same when browsing by both project and student, except for the top level. Other than that the only difference between the two is what is being used as the top-level filter (Student vs. Project).

The first three levels each use the same template as their interface. This consists of a tabbed display which allows the user to choose between a table and a chart. The tabular view presents a table containing data pertaining to each project, user, assignment or submission. The fields available to be shown at each level include:

- Information about the specific data type such as: user name, assignment start and end date or project description.
- Calculated values related to the number of other data types found within this data type. Some examples of this are:
  - The total number of source files in a submission.
  - The average number of testcases submitted per submission for an assignment.
  - The average number of submissions per assignment for a project.
- A submission's final tool results. Due to all results being calculated on a per submission basis, average values are displayed at levels higher than the submission level.

The user can choose which fields are shown in the table by selecting them from a dropdown list. In Figure 4.1 the tabular view can be seen displaying values at the projects level. In order to proceed to a lower level, the user can simply click on one of the rows in the table. For example, clicking on a row representing a specific assignment takes the user to that assignment's submissions table.

Figure 4.2 displays the chart view when viewed at the users level. The chart itself is a scatter plot which compares the various metrics found in the tabular view against each other. It is very customisable allowing the user to plot any of the metrics on either the x- or the y-axis. The user selects which data should be fitted to which axis from a dropdown list and thereafter the chart is dynamically reloaded. Specific information regarding each data point

Figure 4.1: Tabular view of the Projects Level

6 table fields selected

Name	Description	Assignments → Total	Submissions → Total	Source → Total	Passed → Average
TriType		2	17	324 files	92.03 %
KSelect		2	31	2490 files	55.23 %
Watersheds			7	1154 files	48.49 %
Triangle			40	2609 files	41.07 %
Welcome			7	1297 files	21.94 %
BoundedBuffer			16	384 files	N/A
Problematic			8	175 files	N/A
oldclassic		2	8	186 files	N/A

Given a text string, Welcome must determine how many times the string "welcome to code jam" appears as a sub-sequence of that string. In other words, it must find a sequence s of increasing indices into the input string w such that the concatenation of w[s[0]], w[s[1]], ..., w[s[18]] is the string "welcome to code jam".

on the chart can be viewed by hovering over it. The user can proceed to a lower level by clicking on a data point. For example, selecting a specific project loads the chart for that project's assignments. On this chart, the mean value is indicated by the green dot and the two large blue ellipses show where the first and second standard deviations lie. The user can choose to show this additional information and alter the chart in other ways by clicking on the blue "Settings" button. This brings up a settings dialog which can be seen in Figure 4.3.

Both of the chart and table views also allow the user to change the data's granularity without filtering it. For example, suppose the user is at the projects level and they wish to change the granularity to individual submissions. However, they do not want to filter the data to a specific assignment, but rather view all submissions within Impendulo. The user is then able to change the current data's granularity from a dropdown list which reloads the data at the new granularity (Figure 4.4).

The files level (Figure 4.5) also uses the same table template to present information to the user, but it does not provide a chart. Typically the files consist of:

- A primary source file. This file should implement the functionality required by the assignment.
- Additional source files used by the primary file.
- User test files.

At the snapshots level, the data produced by a tool when run on snapshots is presented to the user. This view typically consists of a visualisation in the form of timeline chart as well as reports produced by the tool. The chart is displayed at the top of the view with the reports arranged below it.

In Figure 4.6 we can see that the timeline chart plots the tool's results for all of the snapshots on a time chart. The currently selected snapshots are highlighted and additional information about each snapshot can be viewed by hovering over it on the chart. The time used by the chart is the elapsed time from when the submission was created to when the snapshot was recorded. This is plotted against values provided by a tool's result.

Each result can provide a number of different values to plot on the chart. Furthermore, results from other submissions can also be plotted on the same chart. Lastly, results related to the submission, such as results of another test, can also be added to the chart. These results can be added to the chart by loading them from a dropdown list. After being loaded, the values which the user wants to be shown on the chart can be chosen via checkboxes on the chart's legend.

Below the chart there are two reports displayed side-by-side. The layout of a report is determined by its tool. This layout is loaded from the HTML code the tool is required to provide for rendering its results.

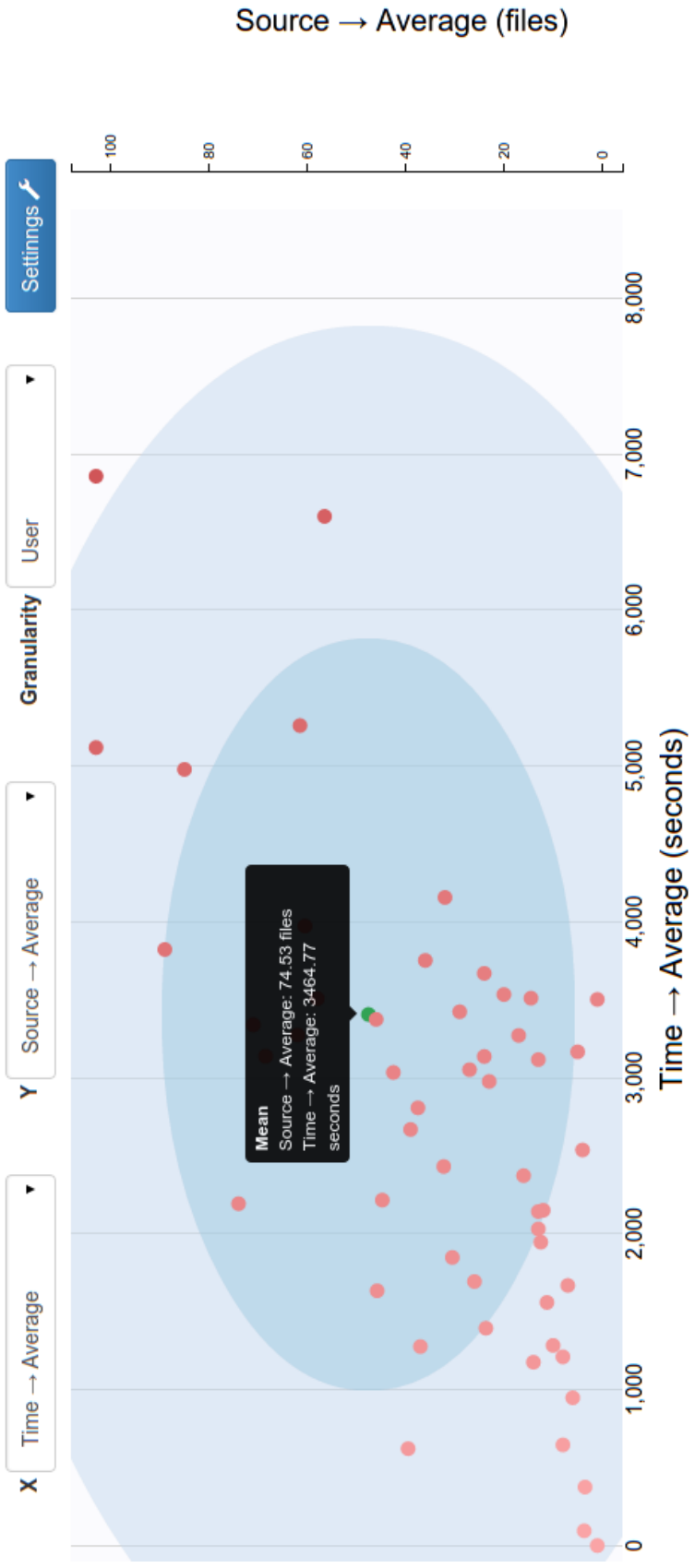
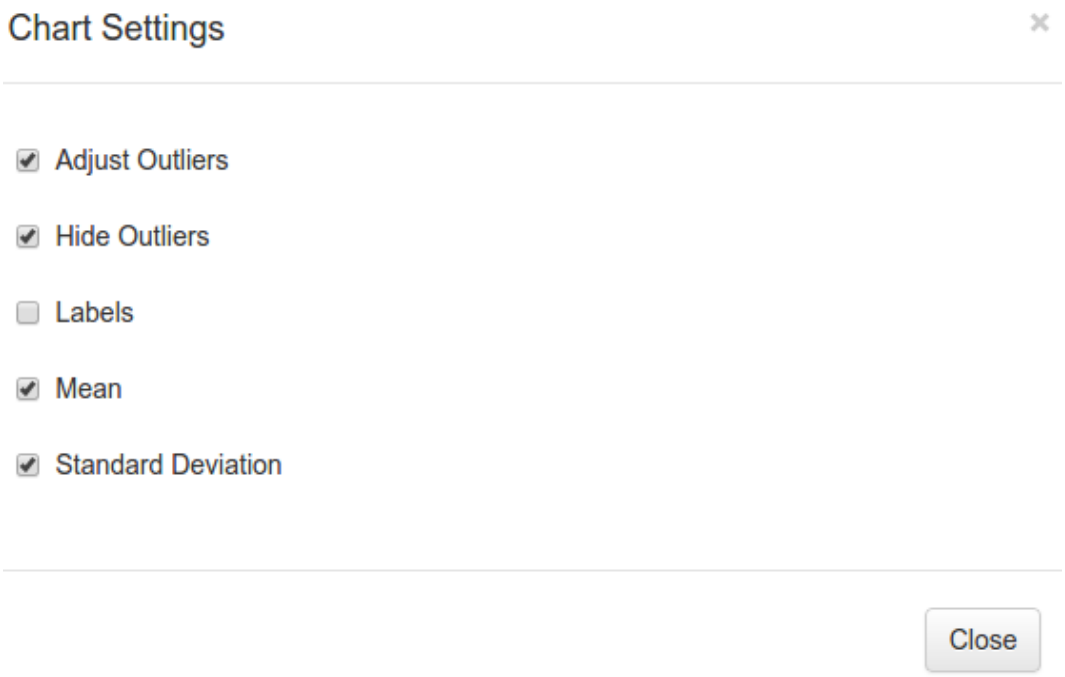


Figure 4.2: Chart view of the Users Level

Figure 4.3: Chart settings dialog



This setup allows us to compare the reports of any two snapshots of a given file. Typically the reports displayed are of consecutive snapshots of a given file. It is also possible to compare any two of the file's snapshots by selecting them from a dropdown list. Furthermore one can iterate through consecutive snapshots by selecting them from a pager widget. The reports provided by the tool's results are specific to the tool but contain all the data seen in the chart as well as a more detailed description thereof. If a report specifies a line at which an error or warning occurs, the tool can be configured to display a popup of the location in the code where this occurs.

Besides analysis tools, the user can also choose to view each snapshot's source code and the differences between the two snapshots they are currently comparing. When viewing the source code, there is the option to turn on annotations. Annotations can be warnings and errors supplied by tools or they can be comments supplied by users. Therefore each annotation is associated with a particular tool or is a comment associated with a user.

Each type of annotation can be individually configured so as to make them easily discernible to the user. This is done via a pop-up box (Figure 4.7) which allows the user to:

- Toggle each type of annotation on or off. Multiple annotation types can be active at the same time.
- Toggle categories of annotation within a specific type of annotation on or off.

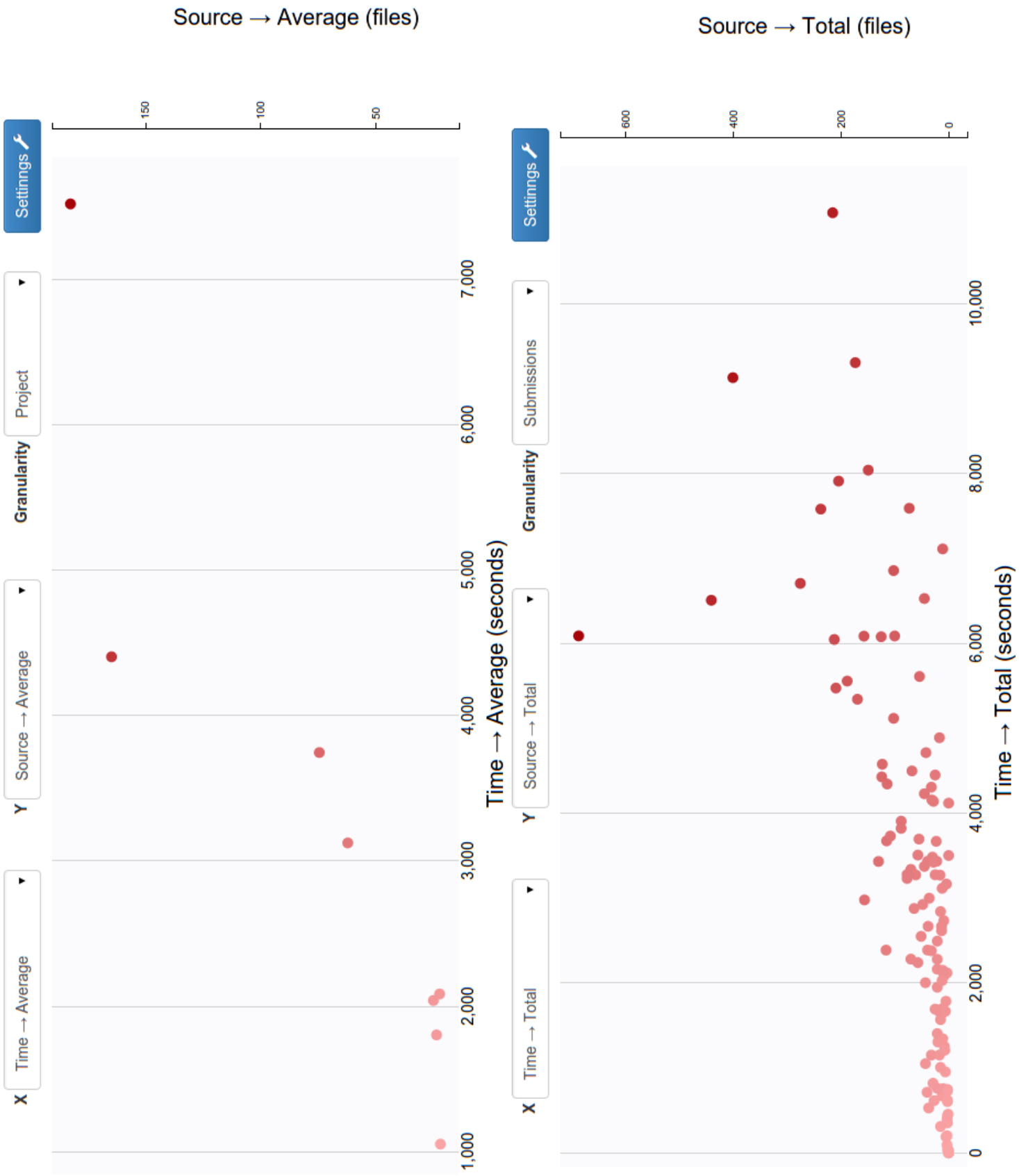


Figure 4.4: Changing the granularity at the projects level.

9 table fields selected

Name	Package	Type	Date	Time	Pmd → Errors	Checkstyle → Errors	Code → Lines	Source → Total
KSelect.java	kselect	Source	2009-10-20	11:25:20	29	223	89	10 files
Pair.java	kselect	Source	2009-10-20	10:56:16	22	143	59	1 files

Figure 4.5: The files level

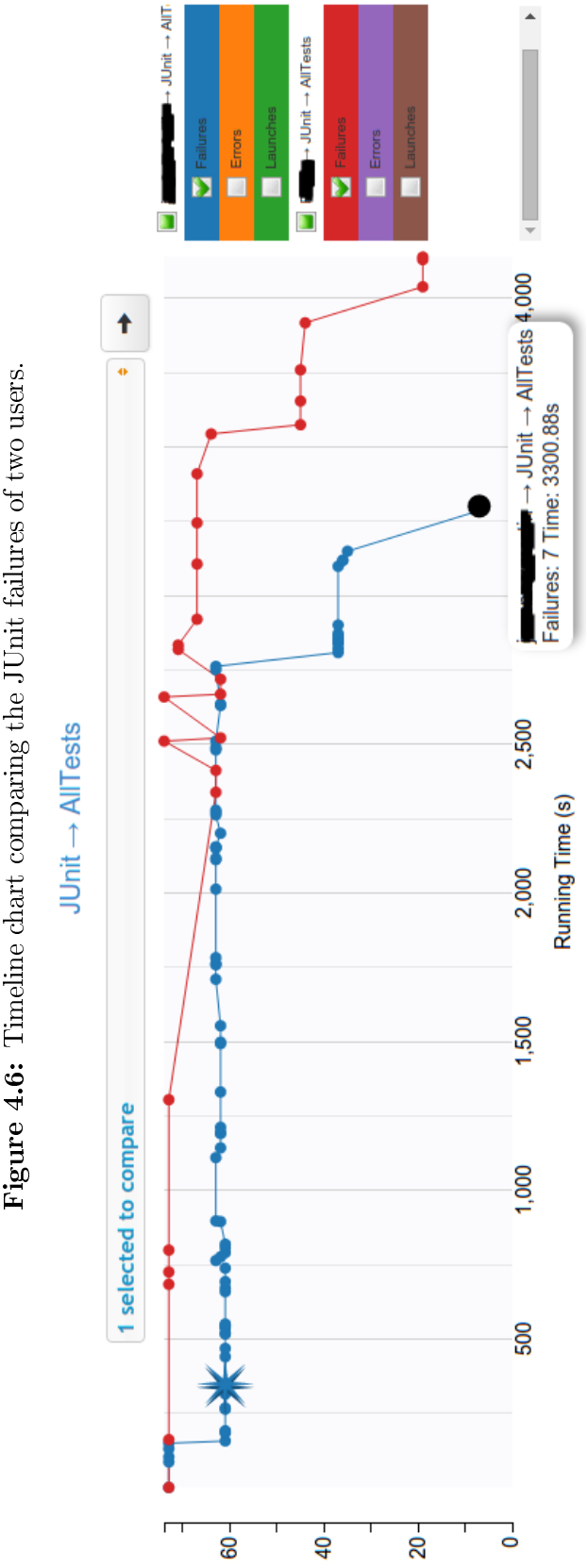
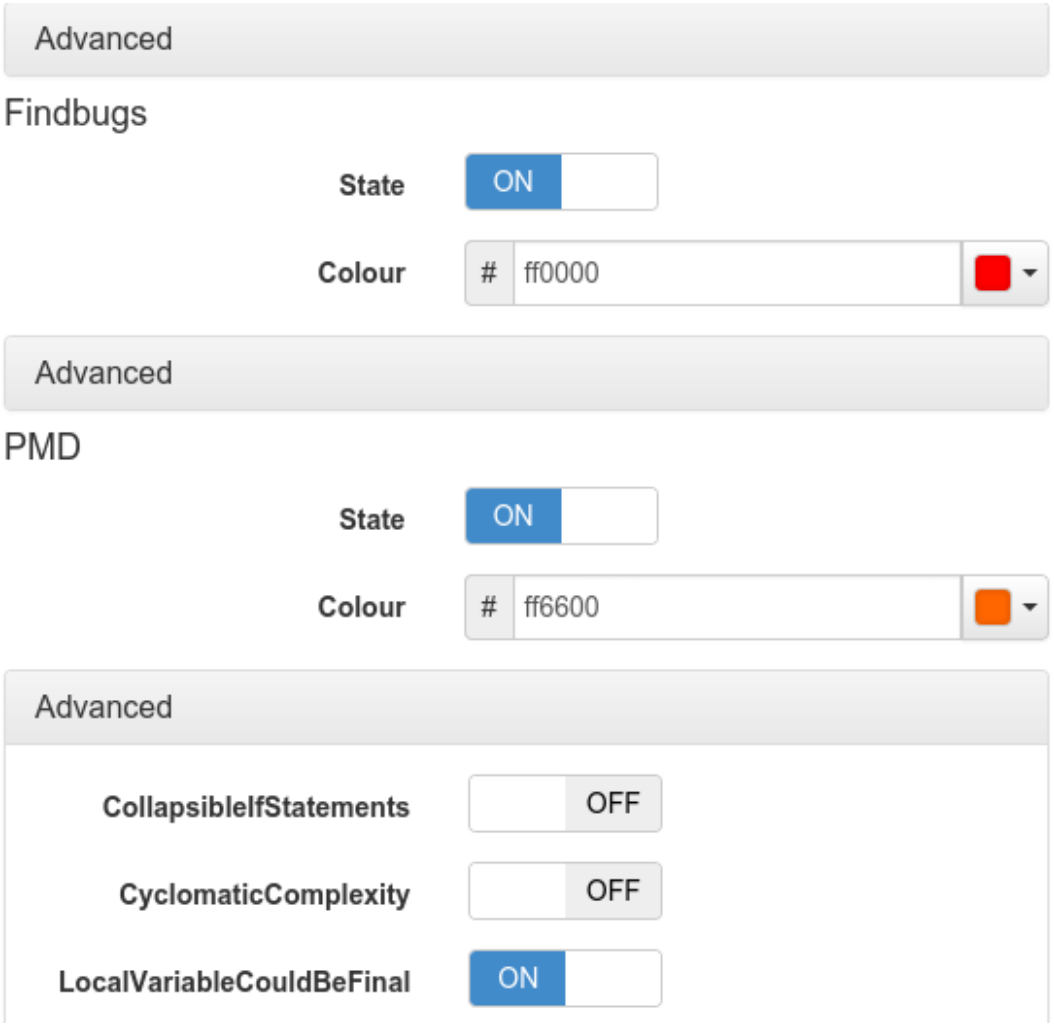


Figure 4.6: Timeline chart comparing the JUnit failures of two users.

Figure 4.7: Configuring Source Code Annotations

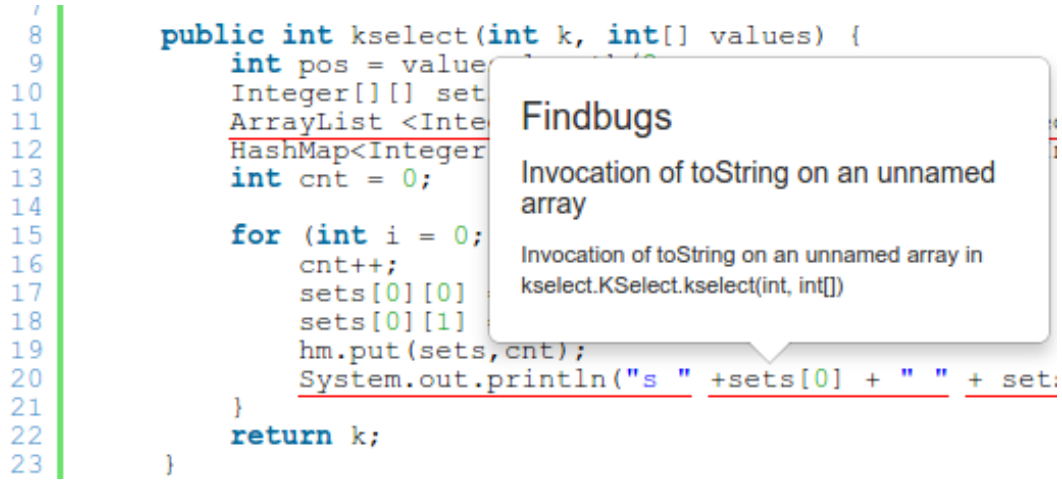


- Set a colour for each annotation type.

Annotations are displayed to the user in two ways. Firstly, the line(s) to which an annotation is adding additional details can be underlined in that annotation type's colour. Secondly, when a user clicks on the line(s) affected by an annotation, a popup appears which displays all the active annotations associated with this line in their respective colours (Figure 4.8).

The user can add comments to specific line(s) in the code by clicking on an edit button next to a line. The user can then write their comment and also set the line(s) with which the comment is associated. The user's name and their comment is displayed whenever these lines are hovered over and comments are active.

Figure 4.8: Source Code Annotations



4.2.4.2 Administration Interface

The Administration Interface is used to manage the data used and produced by Impendulo. This includes everything from creating a new project to editing a test file.

We firstly look at user creation. This is done by new users themselves via the registration page. It is a simple process in which the user simply has to choose a user name and a password. After it is verified that the user name is not already in use, the user is registered and logged in. If the user was already registered, they can simply log in to their existing account.

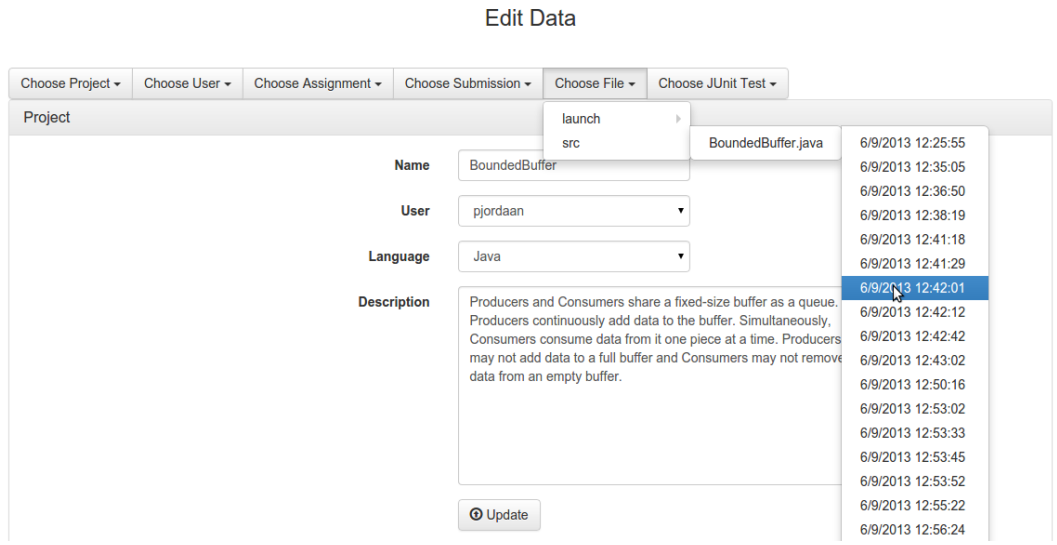
Next we have project creation. This is only available to users with a privilege level of teacher or higher. A new project requires a name, programming language and description. Additionally, a project may have skeletons, assignments and tool configurations. Submitting a new skeleton requires that the user submit a file containing the skeleton's structure as well as its name and the project to which it belongs. Assignments require the user to specify an assignment start and end date, an associated project and a skeleton.

An example of a project's tool configuration is a *rule set*. Rule sets are used to determine which checks a static analysis tool should use when being run.

Currently, the type of configuration most commonly created for a project are unit tests which can be used by both a unit testing framework and a code coverage tool. Creating a new test requires the user to submit the following:

- The test's project.
- The type of test this is.
- The target on which its test cases are run. In Java this is a class file's executable name such as `java.util.HashMap`.

Figure 4.9: The data modification interface.



- The actual test file containing the test cases.
- Any data files required by the test cases. These are submitted as a zip file.

It is also possible to edit some of the data used by Impendulo and in Figure 4.9 we can see the primary data modification interface. It allows the user to alter projects, assignments, submissions, files, users and tests and consists of a series of dropdown lists from which a specific one of these components can be selected to edit. When an item is selected from a dropdown list, its interface for data modification is opened and other open interfaces are minimised.

The Web Application also has a data import and export utility in order to easily migrate data between Impendulo installations. This utility imports and exports data as a zip file containing MongoDB collections with each individual collection stored as a JSON file. The `mongoexport` and `mongoimport` tools are used to achieve this functionality.

The Administration interface also provides instructors with the ability to run specific tools on selected submissions. This is useful when adding new tools or tool configurations to an assignment setup. Any existing results for the specified tools are overwritten.

Lastly, the administration interface allows users to upload Intlola archives for processing. Currently, this submission system only supports zip files.

Chapter 5

Evaluation

In this chapter we describe the experiments we conducted with Impendulo. We detail how each experiment was setup when they were conducted and what it was we wanted to achieve with them.

Next, we take an in-depth look at the results of these experiments. We show how the analysis and visualisation capabilities of Impendulo allow us to identify a user's problem areas.

5.1 Experiment Setup

In order to assess the effectiveness and usability of Impendulo, several experiments were conducted with it. The goal of these experiments was not to discern whether Impendulo as a tool would have any effect on the users, but rather whether it could be useful for identifying problems among users.

The users in each of these experiments were students enrolled in the Computer Science Honours program at Stellenbosch University. They were selected from the year groups of 2009, 2010, 2013 and 2014. The number of students participating in the experiments ranged from 3 to 42. In total 8 different experiments were conducted with each experiment consisting of one or more assignments which the users had to complete.

The assignments given to the students fell into three broad categories:

Basic For these assignments, students are provided with stub methods, a specification and a set of testcases. They are then required to implement the methods so that they fulfill the requirements of the provided specification and pass all the testcases.

Testing As the name suggests, this type of assignment prioritises testing. It has the same attributes as the first type of assignment, apart from way in which testing is handled. Students are provided with fewer or no test cases at all. Additionally, a stub test file is provided in which they must write their own testcases.

Bug fixing This type of assignment provides students with a faulty program and a description of what functionality it is required to exhibit. They are then required to identify and remove all bugs in it. Incidentally, all of the assignments in this category were concurrency related problems.

For a detailed description of each of the projects used for these assignments see Appendix B.

Students were given an hour to complete each assignment in an experiment. This means that if there were three assignments they would be given three hours. However, students were not forced to finish each assignment after an hour. Furthermore, completing all assignments was not prioritised. Lastly, the order in which assignments were to be attempted was also not set in stone. This means that not all assignments had the full quota of students working on them.

The experiments were conducted in the Honours lab of the Computer Science division. All of the assignments were in the Java programming language. Students were required to work on the assignments in the Eclipse IDE and they used the Intlola plugin to submit their work to Impendulo. Students were not provided with or told to use specific usernames and were free to register as whomever they wished. However if they had previously used the Impendulo platform they were encouraged to use their previous account.

Each student was given a step-by-step guide detailing how to install Intlola in Eclipse. They could acquire Intlola either via an update site or by downloading it from Impendulo's Web Application. This guide concluded with an explanation specifying exactly how they should use Intlola to record their work. Furthermore, for each assignment in the experiment, except the bug fixing problems, they were provided with a complete description of how the problem should be solved.

All experiments were executed successfully with minimal setbacks and students reported that they felt the tool was easy to use and integrate into their workflow.

5.2 Overview of Results

In this section we take a look at all the data generated by Impendulo thus far.

In Figure 5.1, snapshots are compared to program executions and both of these are measured per submission for each project. We may ignore the *Watersheds* project because when its submissions were recorded, the system did not capture program launches yet.

The interesting part of this chart is the top left corner where the three bug fixing projects, Problematic, oldclassic and BoundedBuffer, are found. Each of these projects require the user to find and eliminate concurrency related errors from the provided source code. On the chart we can see that the ratios

of program launches to snapshot submissions for these projects are all much higher than for any of the other projects ($\sim$ 0.79 on average compared to $\sim$ 0.14).

This can be due to the non-deterministic nature of these problems where different program executions can produce different results. This means that bugs in these programs are not guaranteed to be visible every time the program is executed. Therefore, users may run their programs many more times than they would a sequential one in order to ensure that they have eliminated all defects.

Furthermore, because the users are not writing a program for these problems they are probably not saving their work as much as if they had to implement an entire method.

Lastly, in order to find the bug initially, users may run the program several times without actually modifying the source code.

Figure 5.2 plots the percentage of testcases passed against the lines of code written per submission for each project. The percentage of testcases passed can be viewed as a measure of the difficulty of the project, however, due to the same students not working on each project and the small sample sizes, this is probably not very accurate.

This chart seems to indicate that students tend to write more code when working on the more difficult problems. When the chart's granularity is changed so that we view all assignments instead, this relationship seems to continue (Figure 5.3).

However, when the chart is set to show all submissions, as in Figure 5.4, we see a very different relationship. This chart indicates that most users' submissions either do very well (pass more than 70% of the tests) or very poorly (pass less than 30% of the tests). Indeed, there are only two submissions which fall outside this range out of the entire 93 submissions. This could mean that there is a certain threshold of tests passed which, when crossed, leads to the user passing most of the tests. However, this phenomenon is probably due to the nature of the problems used for these assignments. That is, these problems normally have one basic solution which results in a large number of testcases being passed and simply requires a few tweaks in order to pass all of them.

Our next example shows how Impendulo can be used to look at the relationship between code coverage and testcases passed, using the same testcases for both. Figure 5.5 plots the line coverage achieved by the Jacoco code coverage tool against the percentage of test cases passed. As can be seen in this small sample, there seems to be some correlation between these two measures. One might be inclined to say that when tests are able to cover a large percentage of a submission's code, the submission probably passed a high percentage of the testcases.

However, the differences between projects' coverage values are not great at all and all projects actually achieved a high line coverage score. Furthermore,

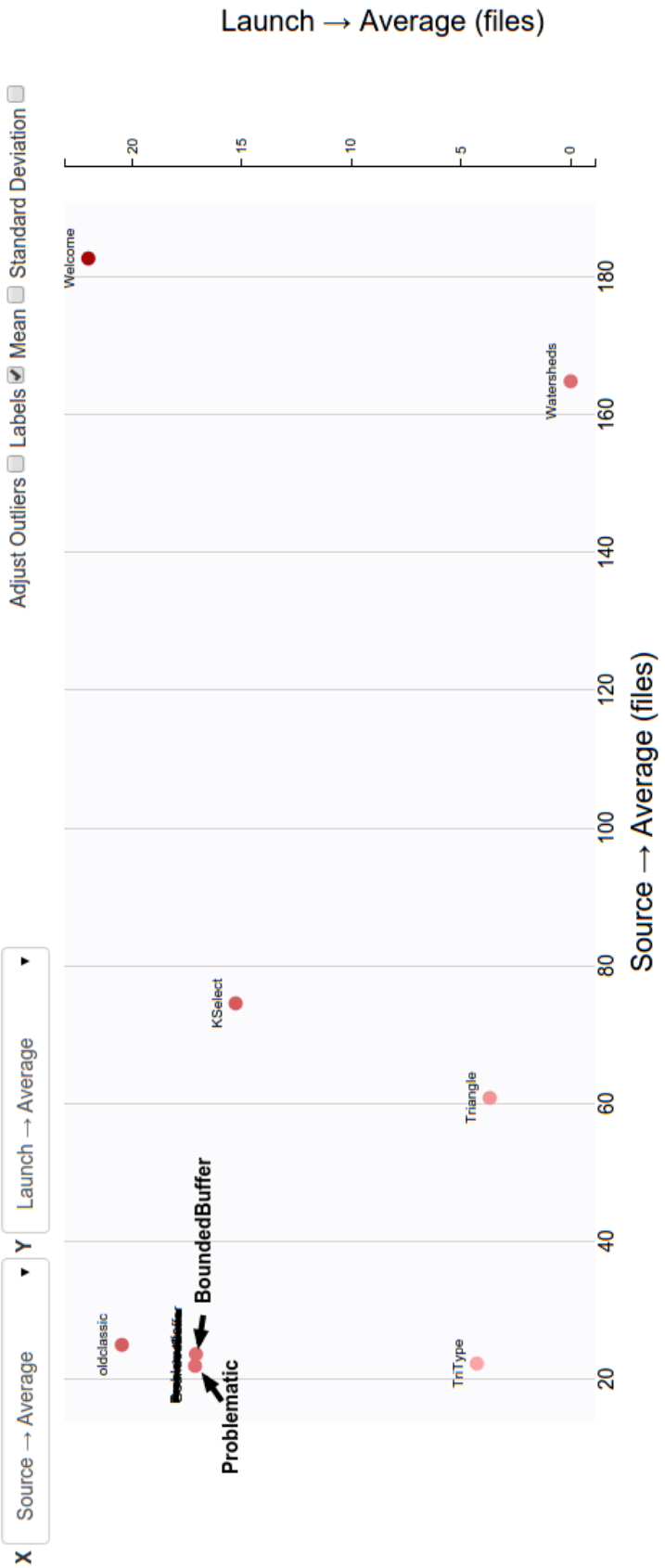


Figure 5.1: Comparison of average number of snapshot submissions and program launches per submission for all projects.

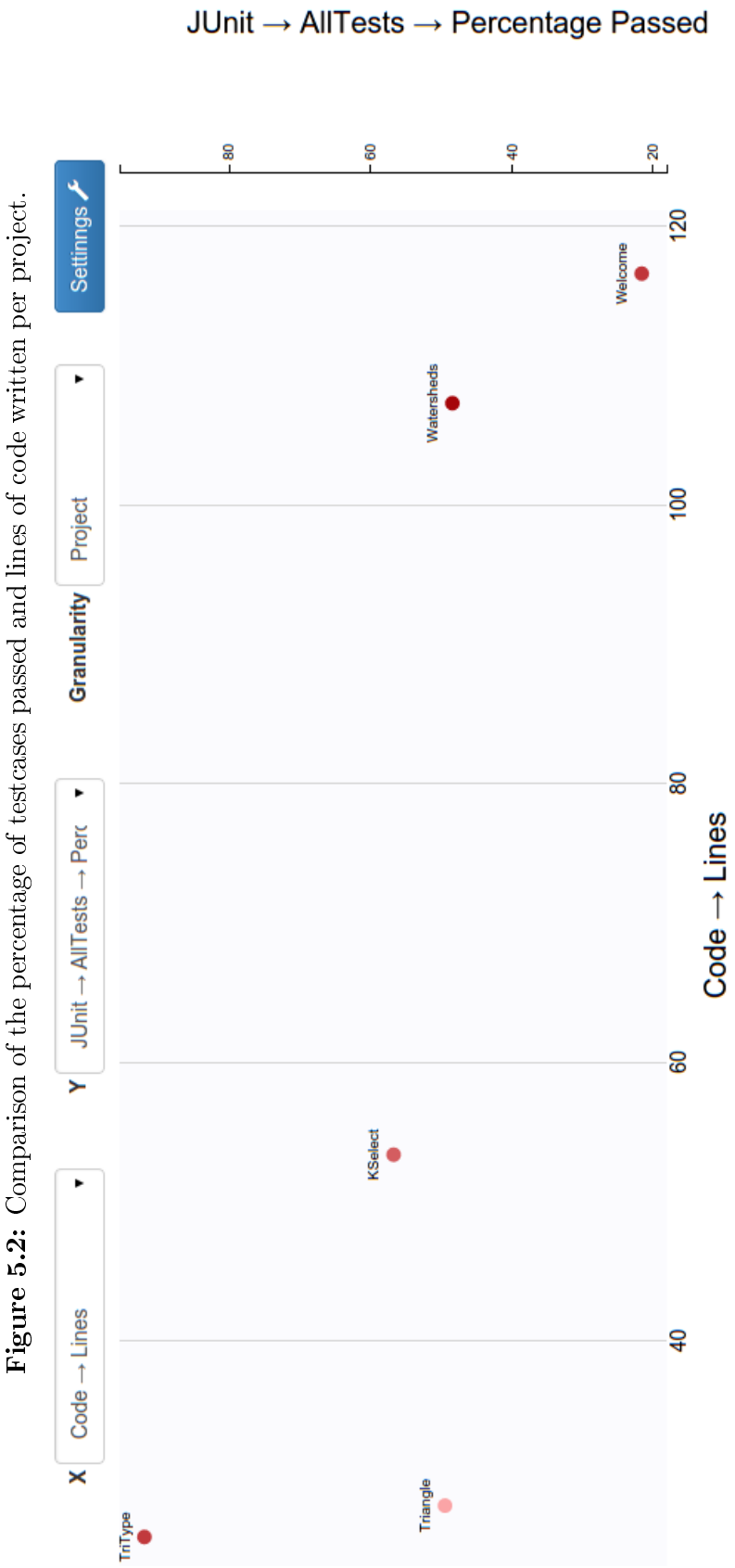


Figure 5.2: Comparison of the percentage of testcases passed and lines of code written per project.

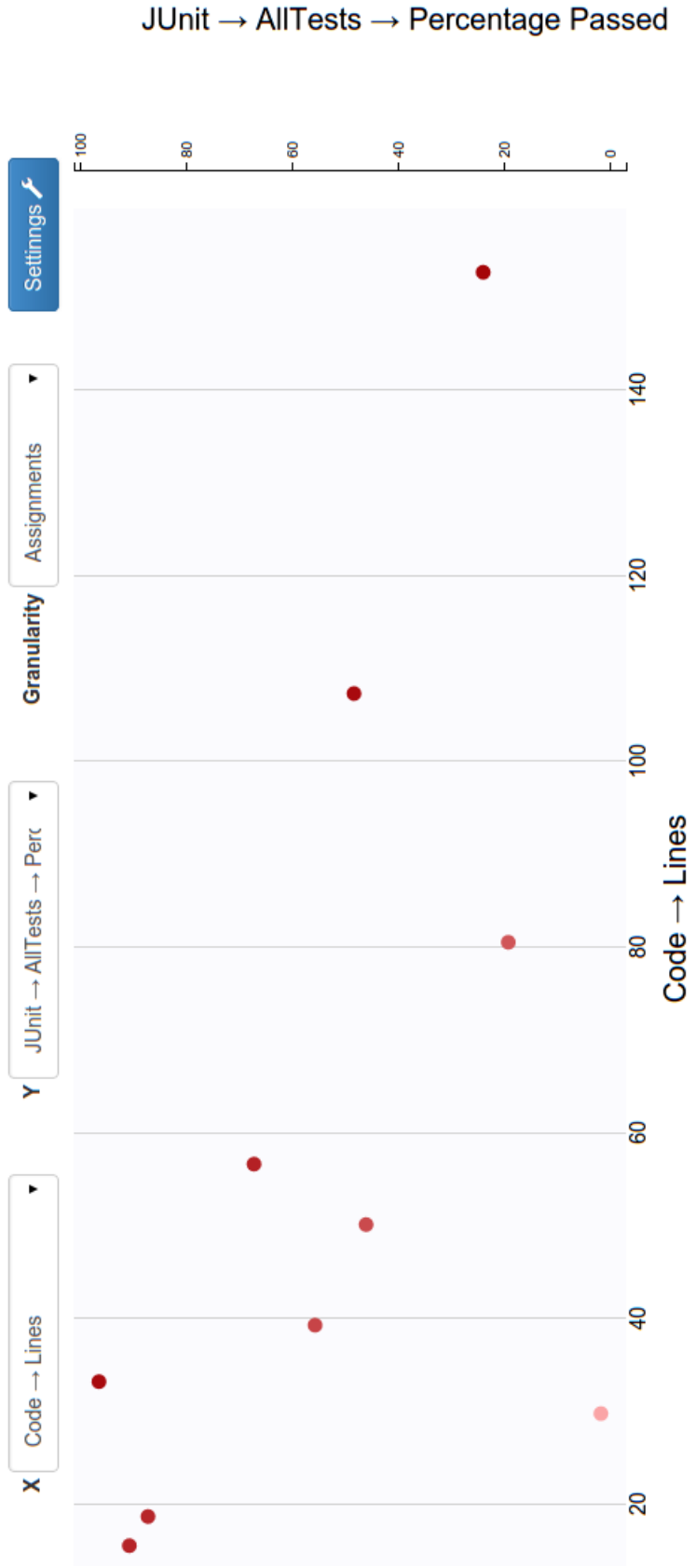


Figure 5.3: Comparison of the percentage of testcases passed and lines of code written per submission for all assignments.

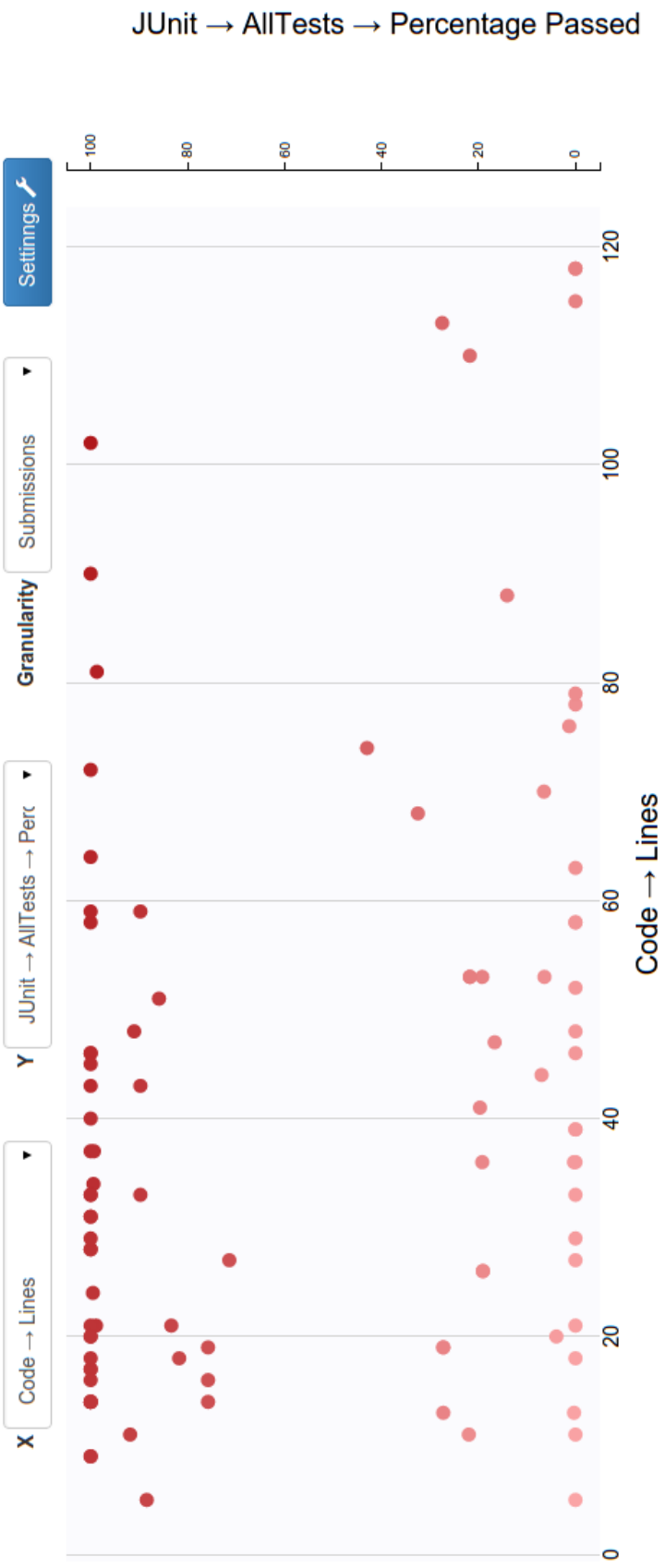


Figure 5.4: Comparison of the percentage of testcases passed and lines of code written per submission for all submissions.

if we change the granularity to display individual submissions, the picture changes drastically.

Figure 5.6 shows that a significant portion of submissions which did get good coverage passed very few tests. In general, most submissions managed to get good line coverage which makes sense given that the problems themselves did not require overly complicated solutions.

Figure 5.7 plots the number of errors detected by Checkstyle against the lines of code written for all submissions. There does seem to be a slight relationship between the two measures, with more lines of code tending to result in more errors. This makes sense to a degree because Checkstyle is primarily a style checker and one would expect more style rules to be broken if there is more code in which to break them.

If we replace Checkstyle with PMD (Figure 5.8) we can still identify the same trend, however, the relationship is not as clear. This can be put down to PMD being a more general static analysis tool and not being as focused on style checking.

In Figure 5.9 PMD and Checkstyle's results are plotted against each other. As one would expect, there is correlation between the two, however, Checkstyle detects far more errors. This could indicate that style violations are more prevalent and/or easier to detect than potential bugs.

The last static analysis tool, Findbugs, does not show this relationship. In Figure 5.10 we can see that Findbugs detected far fewer problems with user code than Checkstyle and PMD. This could be due to Findbugs being a bug detection program and therefore many of the style violations shown earlier do not show up here.

5.3 KSelect

Description The KSelect problem requires the user to compare pairs of integers in order to find a specific pair in a list. Two pairs of integers, (a_1, a_2) and (b_1, b_2), can be compared by first comparing the first component and then the second. Therefore,

$$(a_1, a_2) < (b_1, b_2) \quad \text{if and only if} \quad a_1 < b_1 \quad \text{or} \quad a_1 = b_1 \wedge a_2 < b_2.$$

The user is provided with a list of these pairs from which they must "select" the k-th smallest pair and return its position in the list. When $k < 0$, the task is to find the $-k$-th largest pair. If $k = 0$, or if the absolute value of k is greater than the length of the list, the answer is zero. For example, given the list

$$\underset{1}{(3,1)} \quad \underset{2}{(4,1)} \quad \underset{3}{(5,9)} \quad \underset{4}{(2,6)} \quad \underset{5}{(5,3)} \quad \underset{6}{(5,8)}$$

then

$$(2,6) < (3,1) < (4,1) < (5,3) < (5,8) < (5,9)$$

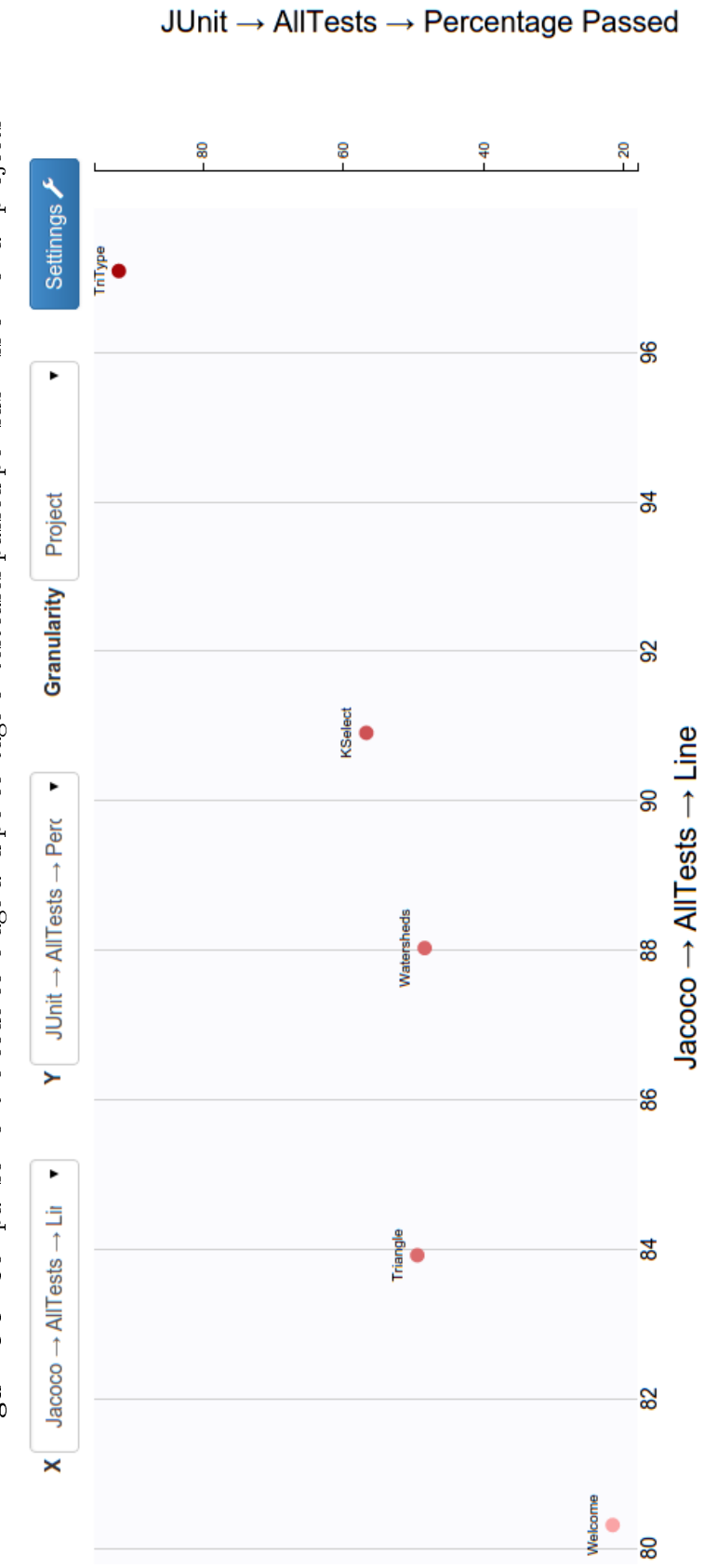


Figure 5.5: Comparison of the code coverage and percentage of testcases passed per submission for all projects.

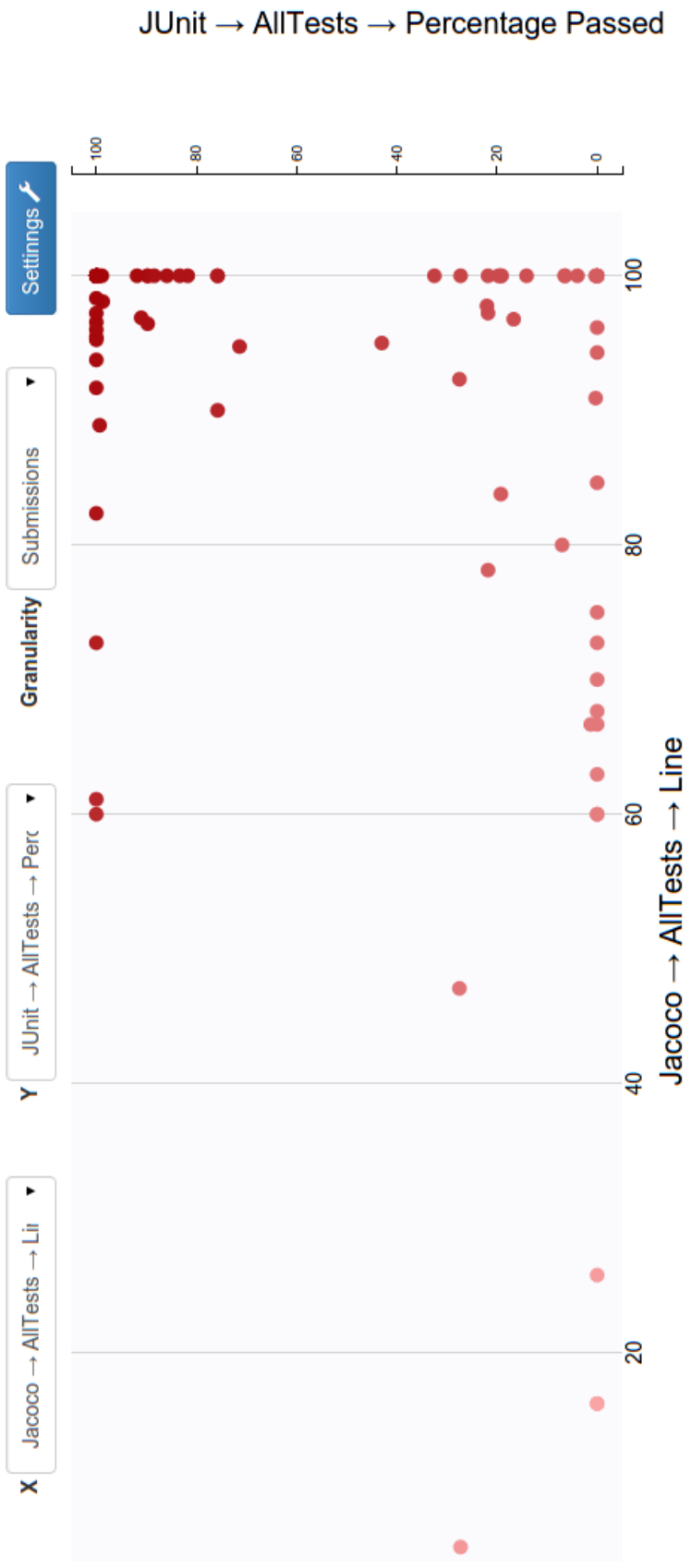


Figure 5.6: Comparison of the code coverage and percentage of testcases passed per submission for all submissions.

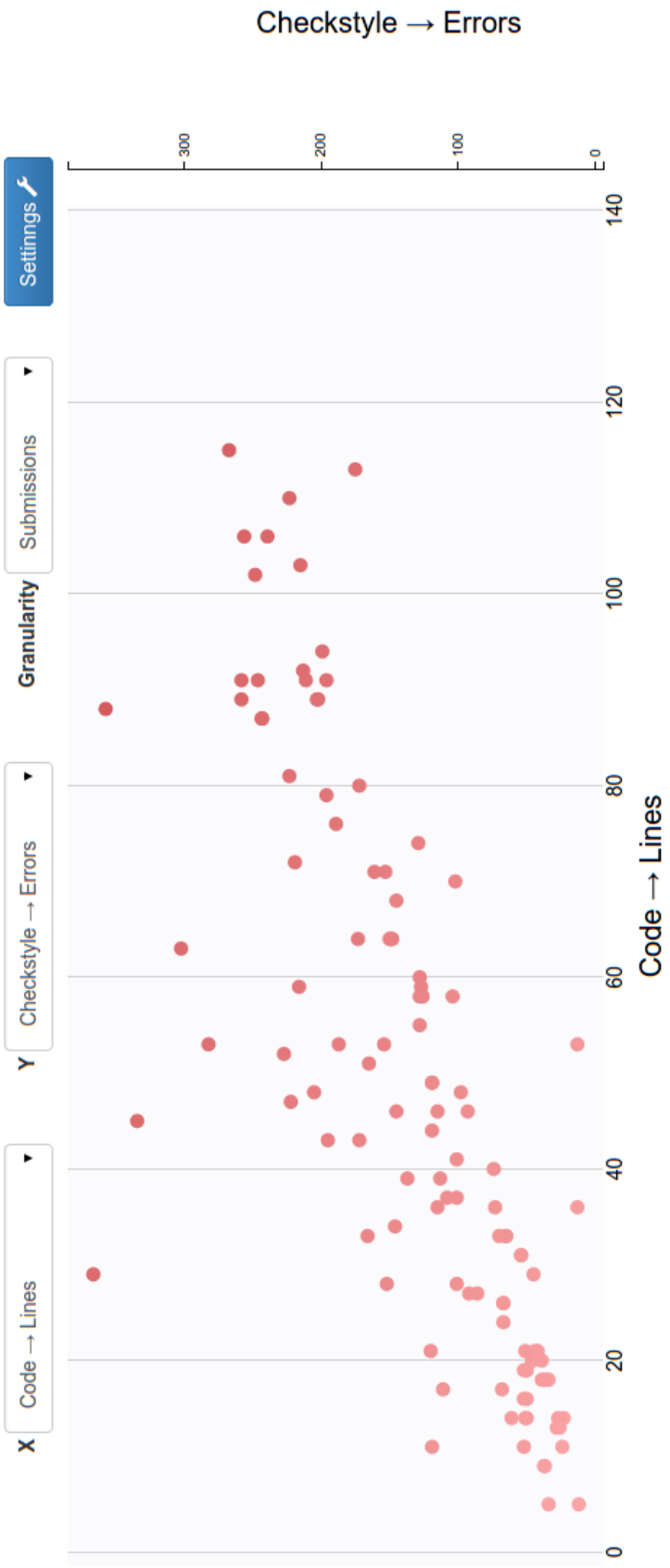


Figure 5.7: Comparison of the Checkstyle errors and lines of code written for all submissions.

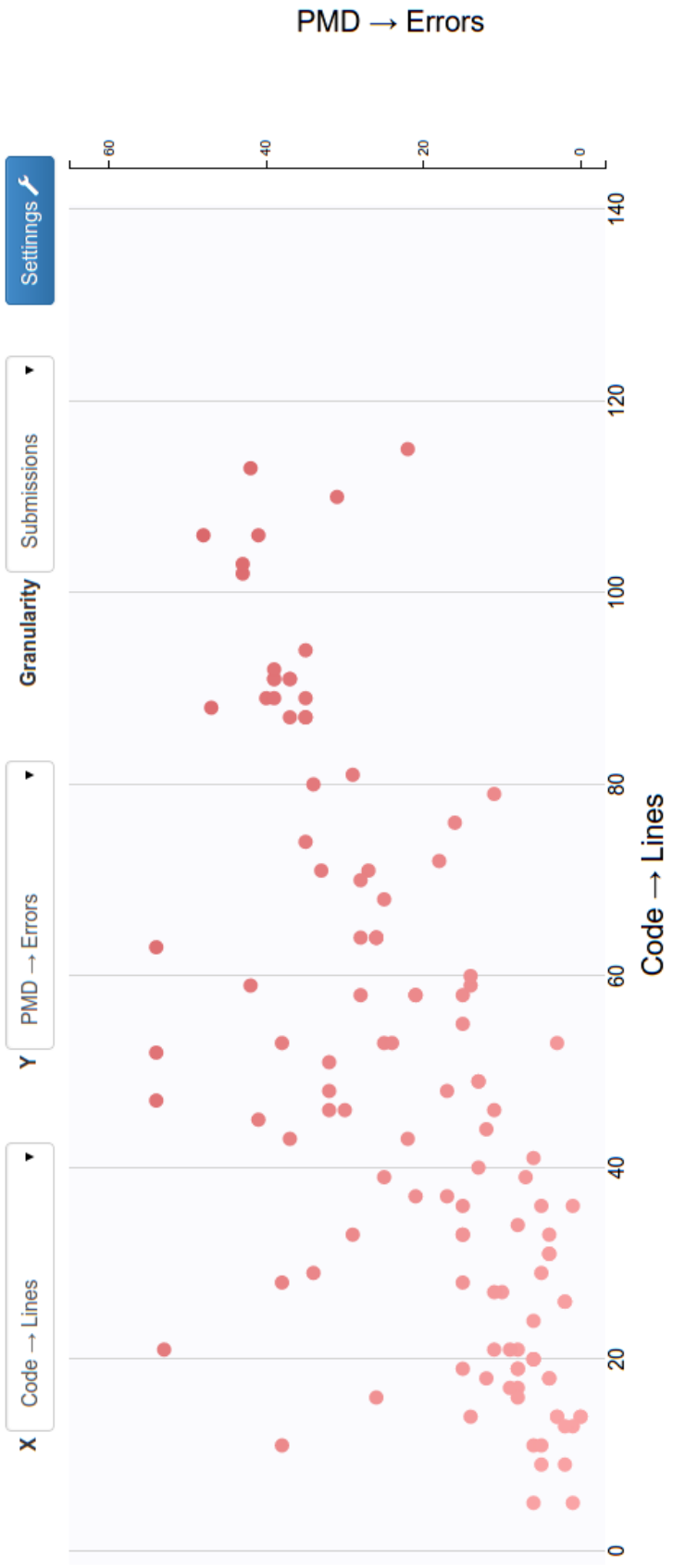


Figure 5.8: Comparison of the PMD errors and lines of code written for all submissions.

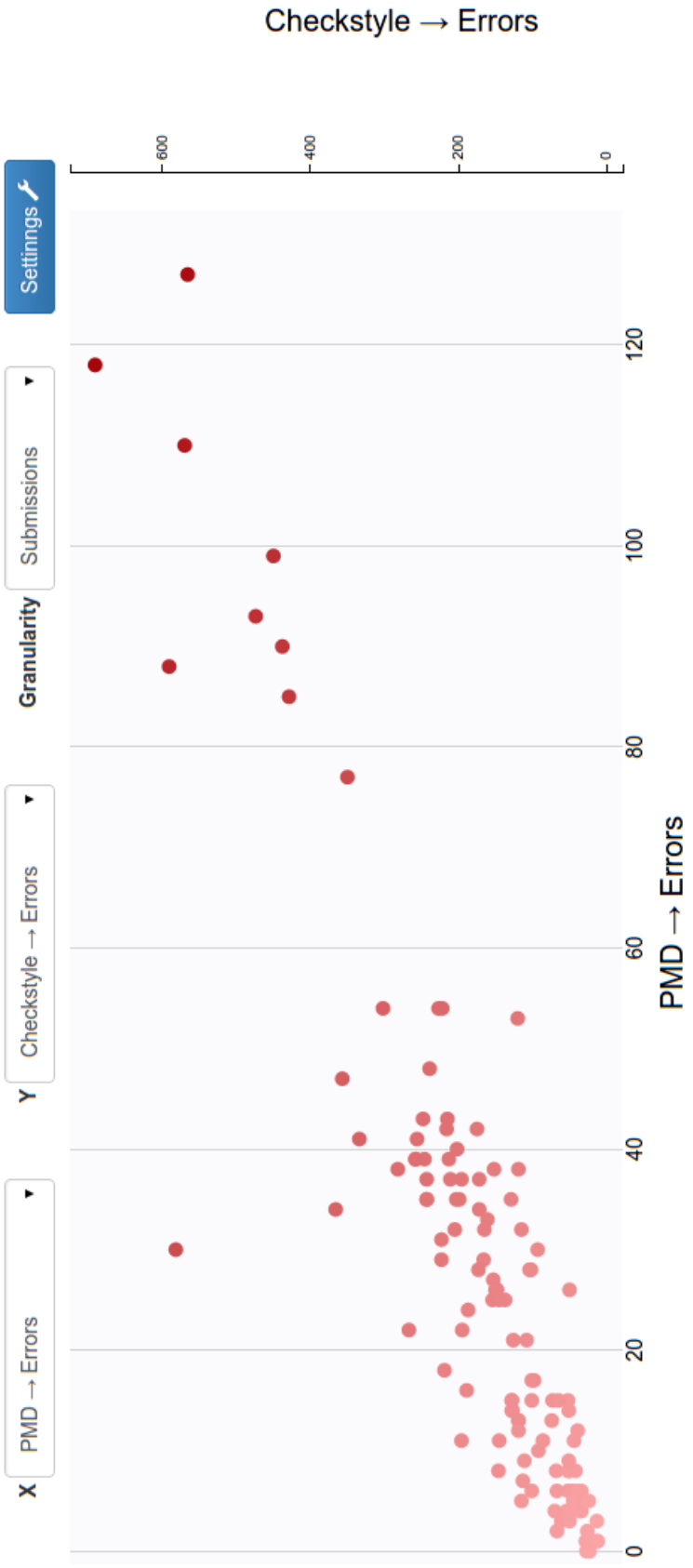


Figure 5.9: Comparison of the Checkstyle errors and PMD errors for all submissions.

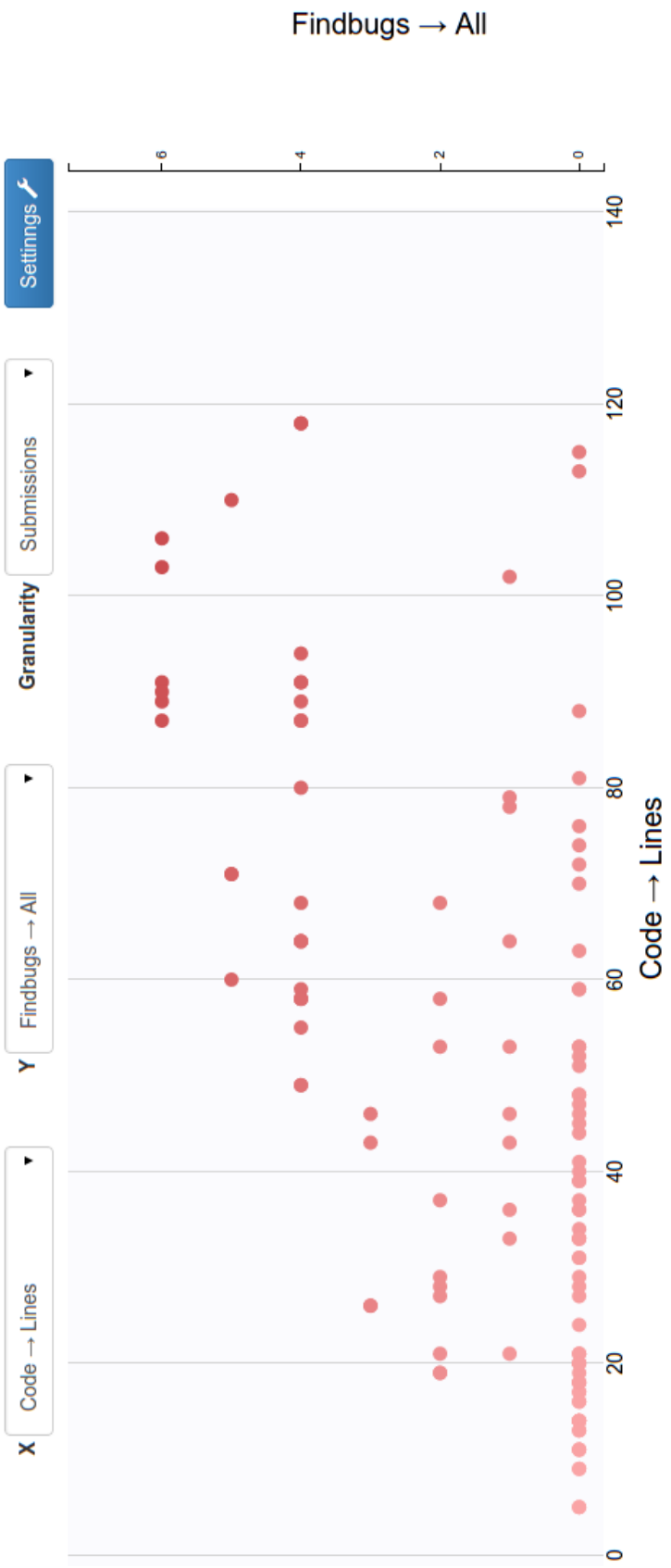


Figure 5.10: Comparison of bugs detected by Findbugs and lines of code written for all submissions.

and we know that:

$$\begin{aligned} 1^{\text{st}} &\text{ smallest pair } (2,6) \text{ is in position } 4, \\ 4^{\text{th}} &\text{ smallest pair } (5,3) \text{ is in position } 5, \\ -1^{\text{th}} &\text{ smallest pair } (5,9) \text{ is in position } 3 \text{ and,} \\ -4^{\text{th}} &\text{ smallest pair } (4,1) \text{ is in position } 2. \end{aligned}$$

The user is required to implement the following function:

```
public int kselect(int k, int values[])
```

The `values` array represents the list of integer pairs with each pair being stored at consecutive indices. For example the array (with indices starting at 0):

3	1	5	4	0	5	12	26	0	3
0	1	2	3	4	5	6	7	8	9

represents the list of pairs (with indices starting at 1):

$(3,1)$	$(5,4)$	$(0,5)$	$(12,26)$	$(0,3)$
1	2	3	4	5

In ascending order this is:

$(0,3)$	$(0,5)$	$(3,1)$	$(5,4)$	$(12,26)$
5	3	1	2	4

Provided with this list, `kselect` should return:

1. 5 if $k = 1$
2. 0 if $k = 6$
3. 2 if $k = -2$

Analysis For this problem there were two assignments:

Honours 20 October 2009 Students were provided with all of the problem's testcases. Furthermore, they were not required to write any tests themselves. We refer to this as assignment A.

Honours 24 February 2014 Students were provided with no testcases and were asked to write tests for their solution. Students' submissions were still tested with the same tests used with assignment A. We refer to this as assignment B.

In Figure 5.11 we can see that as one would expect, assignment A experienced a far higher percentage of passed tests than assignment B. This is due to students working on assignment A having access to the testcases used to evaluate submission correctness and therefore being able to complete their submission knowing that it was correct. What is interesting here however is that the average working time per submission was much higher for assignment A. Furthermore, users working on this assignment wrote 20% more code than those working on assignment B. This can probably be ascribed to the fact that when users were working on assignment A, they were able to constantly check whether their implementation was correct. They were therefore incentivised to continue working until their program passed the tests. This also leads to more code being written with users adding defensive code to weed out bugs.

In Figure 5.12 all the submissions from these assignments are displayed. Here we can see that there are substantial number of users who struggled with the problem (passed less than 50% of the tests). Out of these, the highlighted user's submission stands out since he failed to pass any of the testcases after spending nearly an hour working on the assignment.

In Figure 5.13 we can see that this user struggled to pass any tests throughout the time he spent working on this problem. In fact the most tests he actually passed at the beginning of his assignment when he simply returned `k`. It is interesting to note how closely the errors and failures mirror each other. Here errors are the result of an exception being thrown during the execution of a test. Failures are caused when running a test does not provide the expected result. Most of his changes therefore did not result in a change in the number tests being passed but rather in an increase or decrease in the number of exceptions being thrown.

In Figure 5.14 we can see that JUnit's report of his final implementation is able to pinpoint exactly where one of his errors occurred. Furthermore it shows us that this is a `java.lang.ArrayIndexOutOfBoundsException` which is caused by accessing the array `temp` with the `int` parameter `k` when it is holding a negative value. This indicates that he did not pay attention to the part of the specification regarding negative values for `k`. Furthermore, negative parameter values are normally one of the first things that a user writes tests for when they are implementing a function such as `kselect`. Unsurprisingly, this user did not write any tests.

When the JUnit errors are displayed in the annotated code view (Figure 5.15), we are able to see that they are all `ArrayIndexOutOfBoundsExceptions`. The first occurrence of this issue could be put down to a misunderstanding of the specification. However, the fact that there are two further such errors in his code points to a deeper rooted issue. The importance of bounds checking should therefore be stressed to this student.

6 table fields selected

Name	Submissions → Total	Testcases → Average	Passed → Average	Time → Average	Code → Lines
Honours 20 October 2009	20	0	64.52 %	4102.65 seconds	73.85
Honours 24 February 2014	11	3.64	45.94 %	3387.27 seconds	61.27

Figure 5.11: KSelect assignments: the effect of testcases.

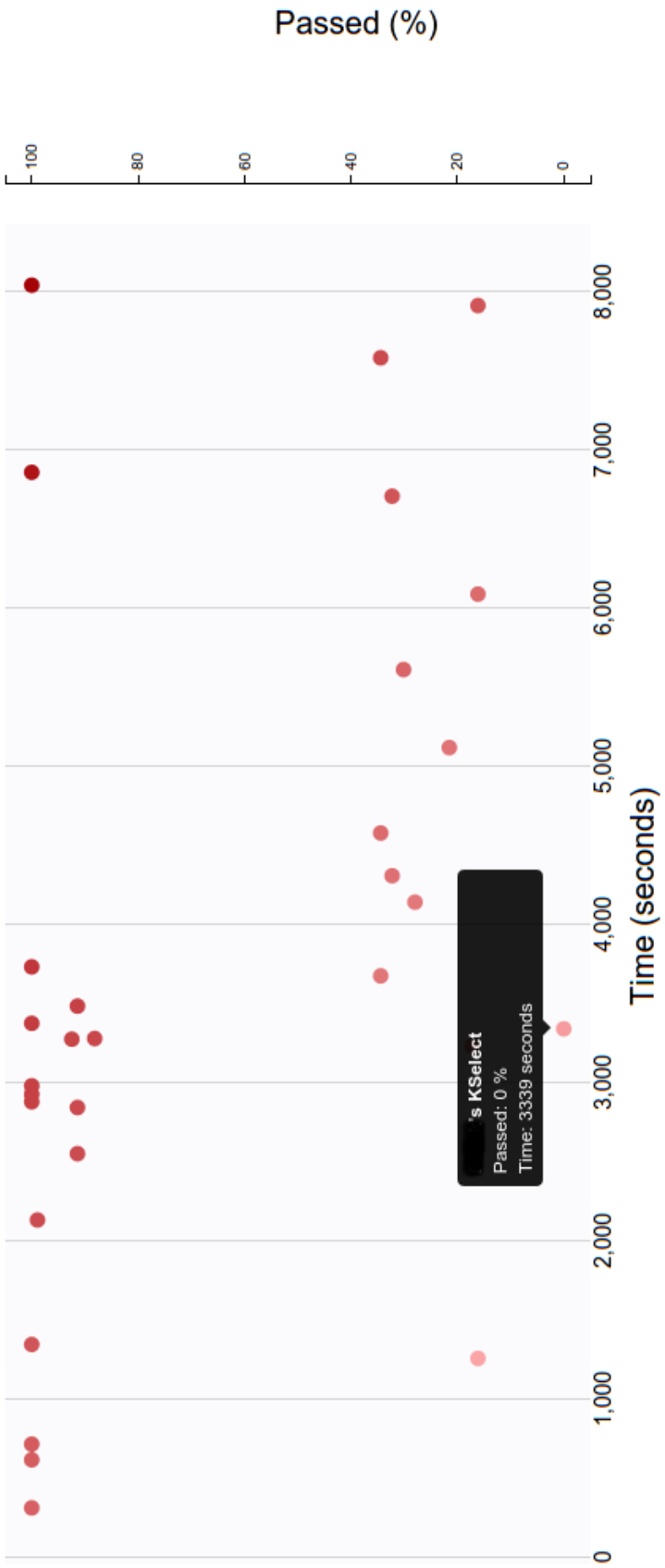


Figure 5.12: Chart depicting user testcases passed versus time spent working on KSelect.

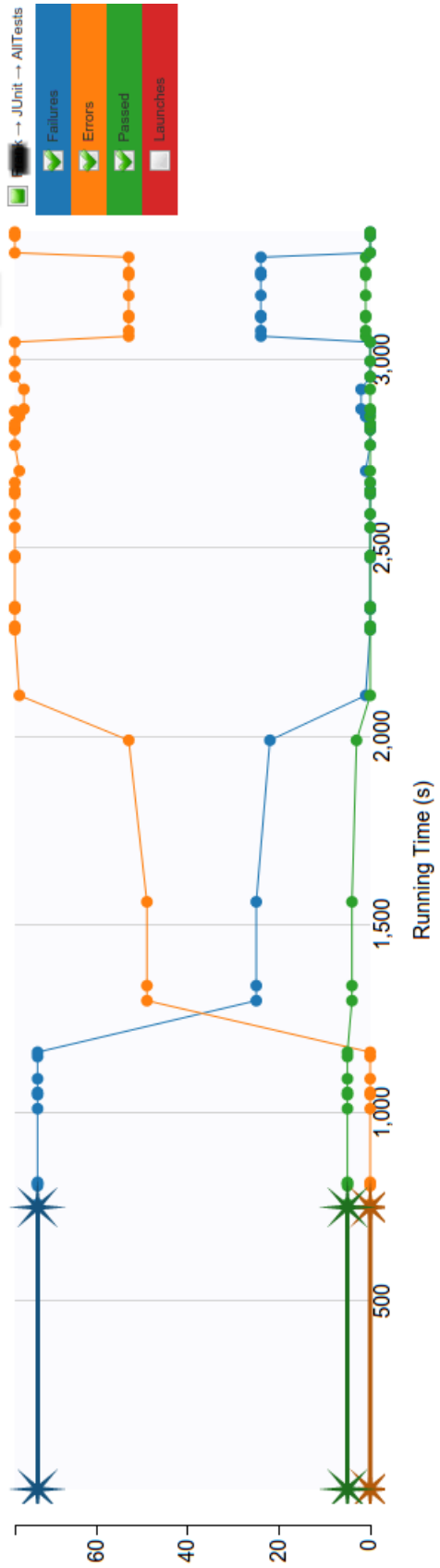


Figure 5.13: A timeline chart of JUnit errors and failures for user X's KSelect submission.

Figure 5.14: A JUnit error highlighted in the source code.

5.4 TriType

Description The TriType problem requires the students to implement the following method:

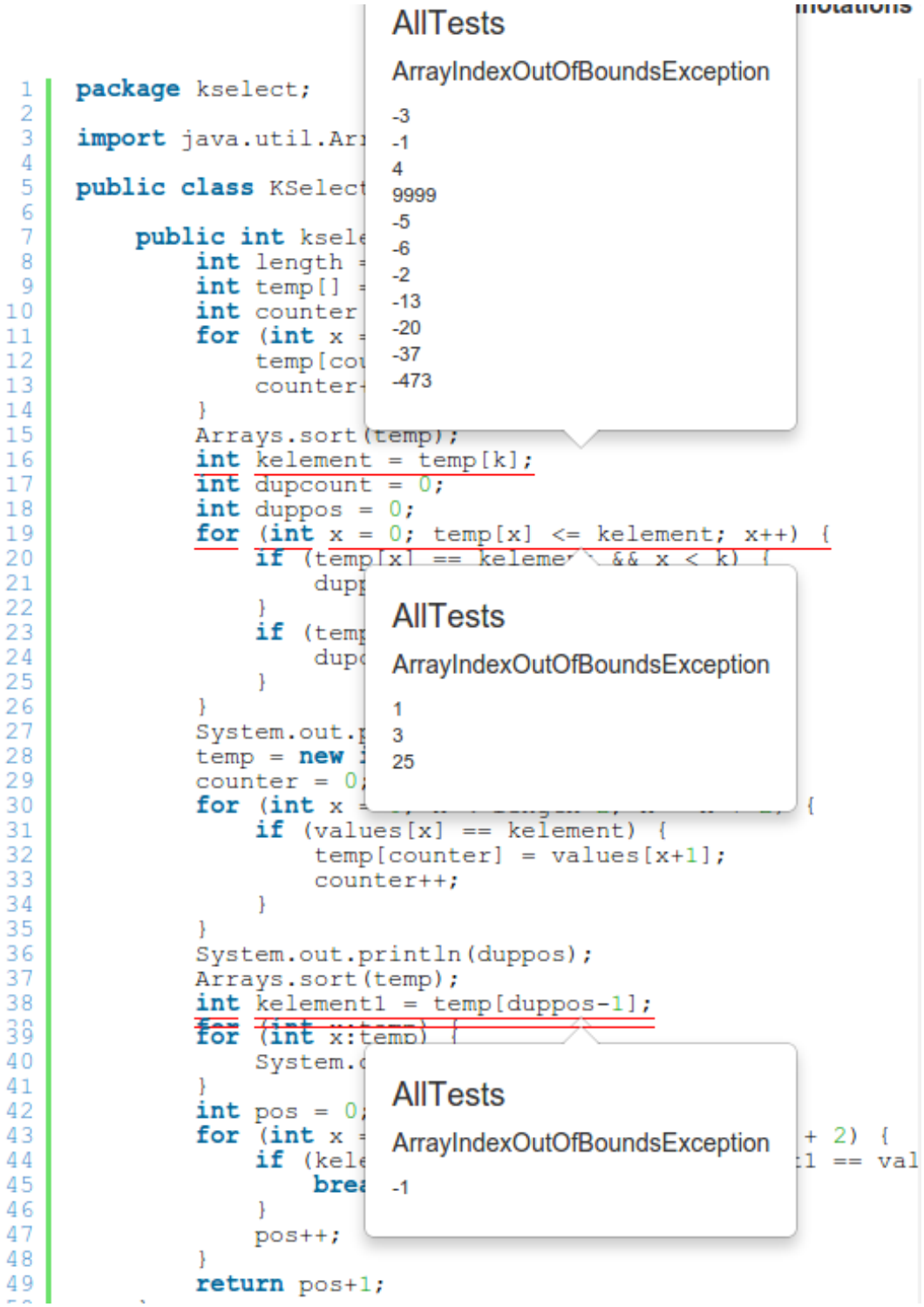
```
public int classify(int a, int b, int c)
```

The parameters each represent the length of one side of a triangle. The method must determine what type of triangle they represent. This is returned as an `int` with the types being:

1	equilateral
2	isosceles
3	scalene
4	invalid triangle

Analysis In Figure 5.16 we can see that once more there are two assignments, one providing the users with testcases and the other requiring users to write their own. Both assignments were evaluated using the testcases given to students working on the first assignment. Similarly to KSelect, users working on the assignment without tests fared significantly worse than those provided

Figure 5.15: Source code annotated with JUnit errors.



with tests. However, the difference is not as great as with KSelect. This could be due to the difficulty of the problem: TriType seems to be more straightforward and easier to solve than KSelect. Therefore users are not as reliant on tests to assist them in the solving of the problem. Another reason could be that users wrote more testcases for TriType (4.9) than KSelect (3.6) on average. The TriType problem is such that it is easy to identify a few obvious tests for it: write a test for each type of triangle. This may have encouraged users to write more tests and therefore bugs were less prevalent in their submissions.

For the next part of the study, we examine only the data from the experiment conducted on the 24$^{\text{th}}$ of February 2014. In Figure 5.17, we can see which students did well on the problem as well as how thoroughly each of them tested their solutions. There does not appear to be a correlation between submitting a lot of testcases and solving the problem. This can be attributed to the fact that the problem itself is relatively straightforward when compared to the other projects. Therefore, users are able to not test their solutions thoroughly and still pass a good portion of the tests.

The student highlighted on the chart submitted a fair amount of testcases and passed nearly all of the assignment's testcases. It should therefore be interesting to determine which part of the problem they struggled with.

The JUnit results of their final submission (Figure 5.18) indicate that all of their failures are due to them classifying a triangle as invalid when this is not the case. Furthermore, Figure 5.19 shows us that the failed testcases themselves all use very large values for at least one side of the triangle.

It would stand to reason that these failures are therefore the result of an overflow problem. This is due to the fact that the parameters are all of type `int` which has a maximum size of $2^{31} - 1 = 2147483647$. When an operation inadvertently pushes the value of an `int` beyond this bound, the result overflows to the minimum value of the `int` type ($-2^{31} = -2147483648$).

Listing C.1 show us the tests which the student wrote for their solution. The tests address each of the different types of triangle as well as handling a few instances of invalid triangles. However triangles with very large sides are not considered despite the fact that the specification states that the parameters can range from `Integer.MIN_VALUE` to `Integer.MAX_VALUE`.

5.5 oldclassic

Description The oldclassic problem requires students to identify and fix a concurrency bug in an existing program, `oldclassic.java` (Listing C.2). The bug in the program is a deadlock which occurs as a result of a missed signal (`wait()` is called after `notify()`). It is caused by directly accessing an `Event` objects's `count` field from concurrent clients (`FirstTask`, `SecondTask`), with-

6 table fields selected

Name	Submissions → Total	Testcases → Average	Passed → Average	Time → Average	Code → Lines
Honours 19 August 2013	6	0	96.66 %	3072.33 seconds	51
Honours 24 February 2014	11	4.91	87.39 %	1098.36 seconds	32.45

Figure 5.16: TriType assignment comparison.

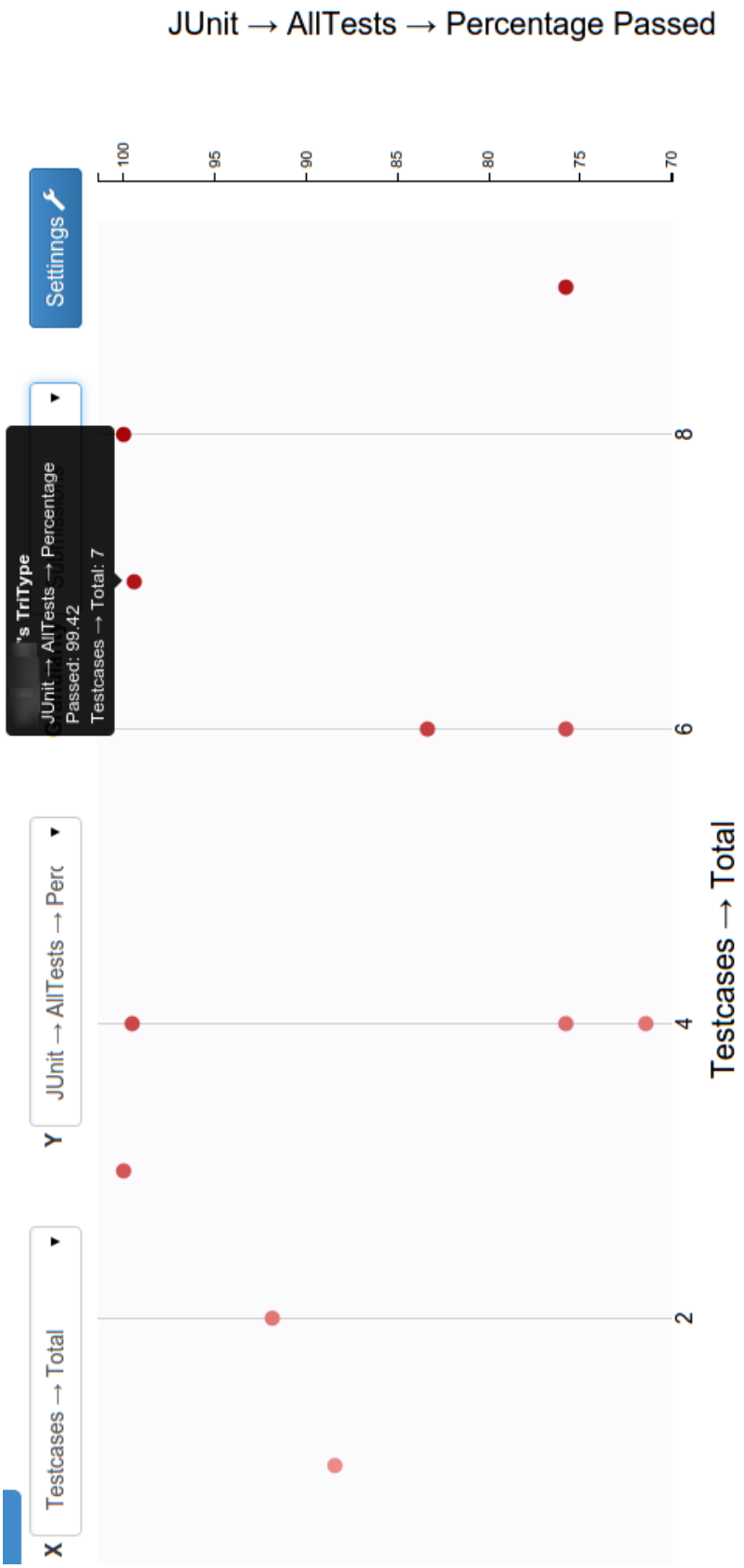


Figure 5.17: Chart of TriType's user testcases submitted versus testcases passed.

Figure 5.18: Incorrect triangle classification.

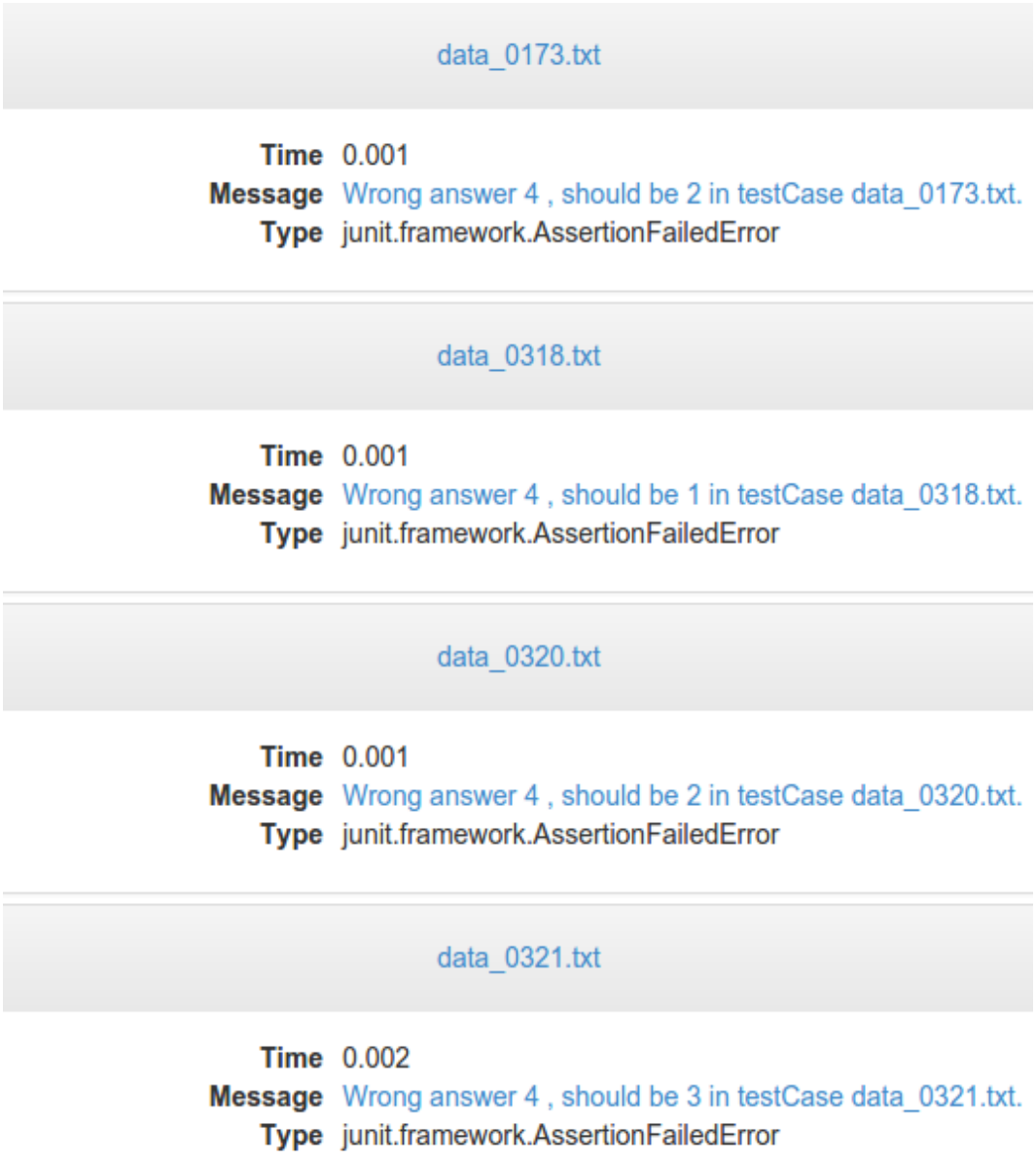
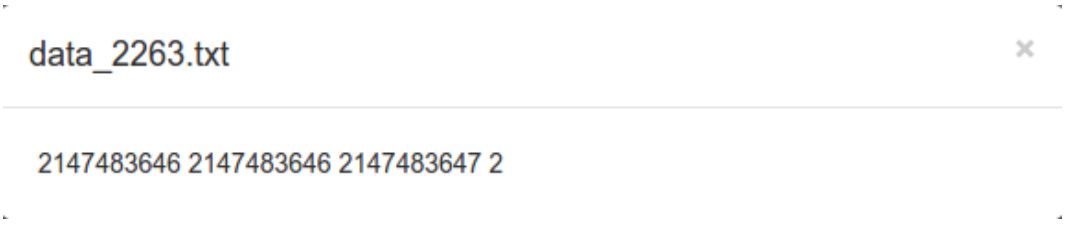


Figure 5.19: Failed testcase.



out synchronization with the corresponding operations (`wait_for_Event()` and `signalEvent()`).

Analysis Now, we inspect a student's approach to identifying and removing the bug. In Figure 5.20, we can see the problems that the JPF tool found when analyzing the student's first and last submissions.

Although the total number of errors which JPF detected decreased, the number of unique errors it found remained the same. This means that the same number of errors are still actually present, they just occur less frequently. The errors found by JPF are three race condition violations and one deadlock. The race conditions probably occurred when both of the task objects attempted to access the `count` field of the same `Event` object.

Findbugs is able to detect a problem with the program as well (Figure 5.21). It alerts us to a synchronization problem when `count` is accessed. Although not as comprehensive as JPF, this is more insight than the other static analysis tools managed to provide.

In Figure 5.22, we can see all the JPF errors simultaneously in the annotated code view. Two of the race condition errors were actually removed by the time the user reached their final submission (Error #1 and Error #2 in the first code listing). However they then introduced two new errors by printing out the `Event` objects's `count` field. This could point to a misunderstanding of the exact cause of the bugs.

5.6 Triangle

Description The Triangle problem provides the student with a triangle of numbers which has the following structure:

$$\begin{array}{cccccc} & & & 3 & & \\ & & 7 & & 4 & \\ & 2 & & 4 & & 6 \\ 8 & & 5 & & 9 & & 3 \end{array}$$

A *path* can be traced through the triangle by starting at the top and moving to an adjacent number on the row below the current row until the last row is reached. For the above example, a valid path is $3 \rightarrow 4 \rightarrow 4 \rightarrow 9$.

The *weight* of the path is the sum of all the numbers visited on the path.

The student's objective is to find the maximum weight of any path in a given triangle. In this example, there is only one path that delivers the maximum weight. The numbers of this path are underlined below:

$$\begin{array}{ccccc} & & \underline{3} & & \\ & \underline{7} & & 4 & \\ 2 & & \underline{4} & & 6 \end{array}$$

Figure 5.20: Java Pathfinder's results for the initial and the final submission.

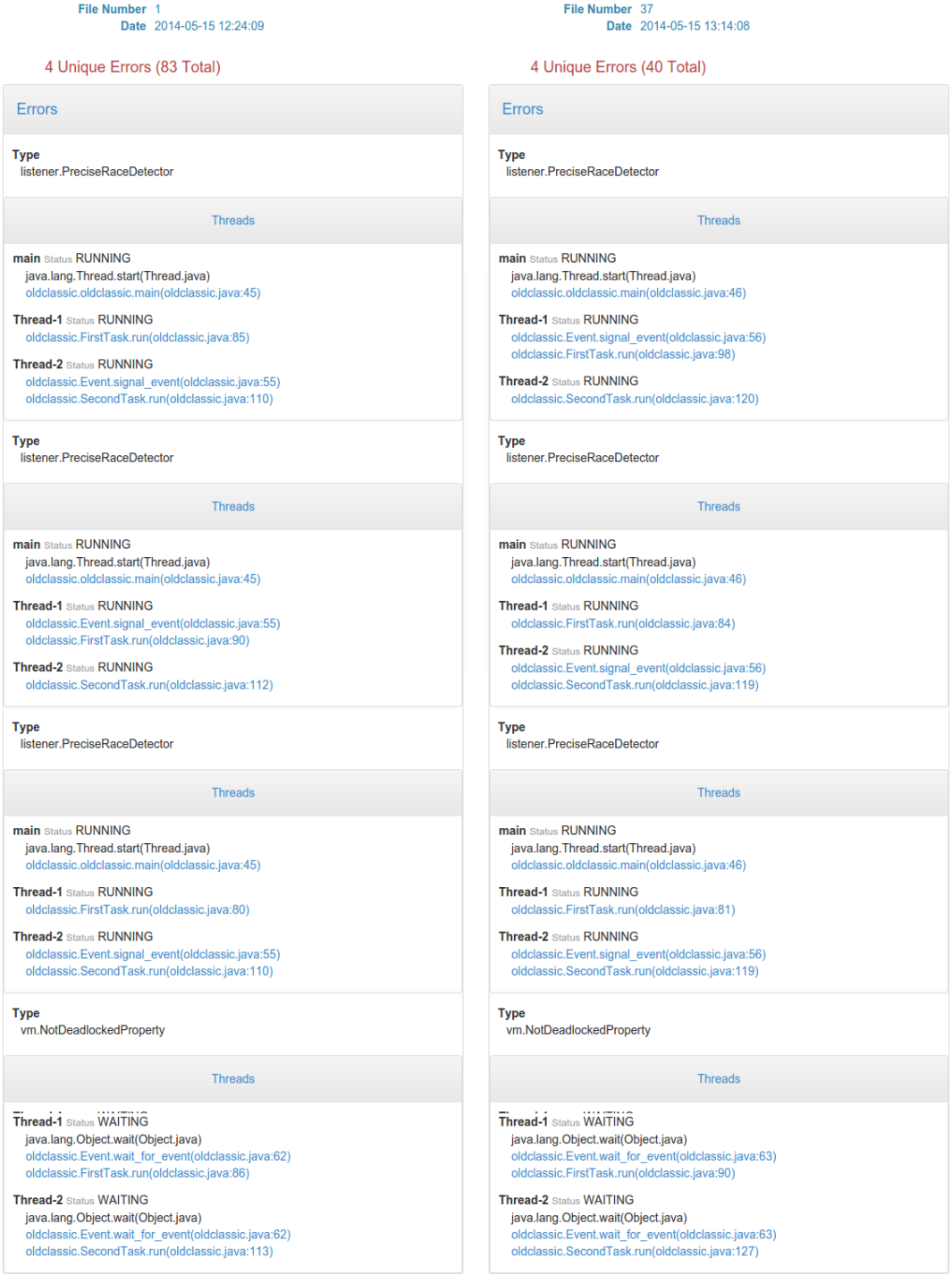


Figure 5.21: A concurrency bug found by Findbugs.

Inconsistent synchronization

Priority	2
Category	Multithreaded correctness
Rank	17
Description	Inconsistent synchronization of oldclassic.Event.count; locked 45% of time
Source Line	Synchronized access at oldclassic.java:[line 88]

8 5 9 3

The path's weight is: $3 + 7 + 4 + 9 = 23$

The student must implement the `maxpath` function which has the following signature:

```
public int maxpath(int tri[][])
```

The two-dimensional array, `tri`, represents the numbers of the triangle. The function must return the weight of the heaviest path as an integer.

The first array element `tri[0]` is a one-dimensional array that contains the top number of the triangle (`tri[0][0]`). The second element `tri[1]` is a one-dimensional array that contains the two numbers in the second row of the triangle (`tri[1][0]` and `tri[1][1]`), and so on.

In other words, the example triangle above is passed to `maxpath` as:

```
{ { 3 },
  { 7, 4 },
  { 2, 4, 6 },
  { 8, 5, 9, 3 } }
```

Analysis In Figure 5.23 we can see that the three assignments conducted for the Triangle problem have widely disparate scores for JUnit testcases. Due to the small size of groups used when conducting the experiments, this can simply be put down to some groups being much stronger than others. However, the pass percentage of only 1.79% achieved by the 2010 experiment seems much too low for a whole group.

Looking at the 2010 experiment in Figure 5.24, we can see that every single student did very poorly. Furthermore, most of them passed 0.3% or less of the testcases. Inspecting each of these solutions individually, however, does not point to any single underlying reason for their poor performance. Three of the submissions' final snapshot either did not compile or timed out when being

Figure 5.22: Java Pathfinder's results annotated in the source code. Initial submission on the left and the final submission on the right.

tested which led to an earlier snapshot's JUnit results being used instead. Another submission just returned the top value of the triangle, however, these cases do not explain why the rest of the users did so poorly and therefore we must simply put it down to being a bad group. This is backed-up by the results of the 2010 Welcome experiment, which featured some of the users from this experiment. In Figure 5.25 we can see that the group did poorly with an average pass percentage of just 19%.

Interestingly, if we plot `EasyTests.java's` JUnit results against those of `AllTests.java` (Figure 5.26), we see that one user managed to pass all of the `EasyTests.java` testcases, but none of `AllTests.java's`. When we inspect this user's results, we see that their last few snapshots timed out for `AllTests.java`, resulting in a much earlier snapshot's results being used. However, `EasyTests.java` did not time out and they managed to pass all its testcases. When we inspect their code, we see that they use a recursive algorithm which has a running time exponential in terms of the number of rows in the triangle. This does not present a problem when it is used with the more simple testcases found in `EasyTests.java`, but `AllTests.java` uses some large triangles which result in the tests not finishing.

5.7 BoundedBuffer

Description BoundedBuffer is an example of the classic Producer/Consumer concurrency problem. The crux of it is that there are two classes of process, a producer and a consumer, which share a common, fixed-size buffer as a queue. Each producer continuously generates data and puts it into the buffer. Simultaneously, each consumer retrieves one piece of data from the same buffer and consumes it. However, producers must not add data to a full buffer and consumers must not try to remove data from an empty buffer.

Students are provided with a version of this program which contains a deadlock due to the wrong thread receiving a signal to stop waiting. This can be fixed by using `notifyAll()` instead of `notify()` to wake the threads since it ensures that the intended thread receives the signal by sending the signal to all waiting threads.

The code for this problem can be seen in Listing C.3.

Analysis Figure 5.27 shows us that the submissions for this problem either completely solved it or did not manage to at all. This agrees with the fact that the solution is a simple one-line change to the source code. Furthermore, the chart seems to indicate that users did quite well on this problem with an impressive 71% of them managing to solve it, this despite them being unfamiliar with Java concurrency programming.

However, if we inspect each submission individually, we see that this is not really the case. Firstly, three users modified the code so that threads would

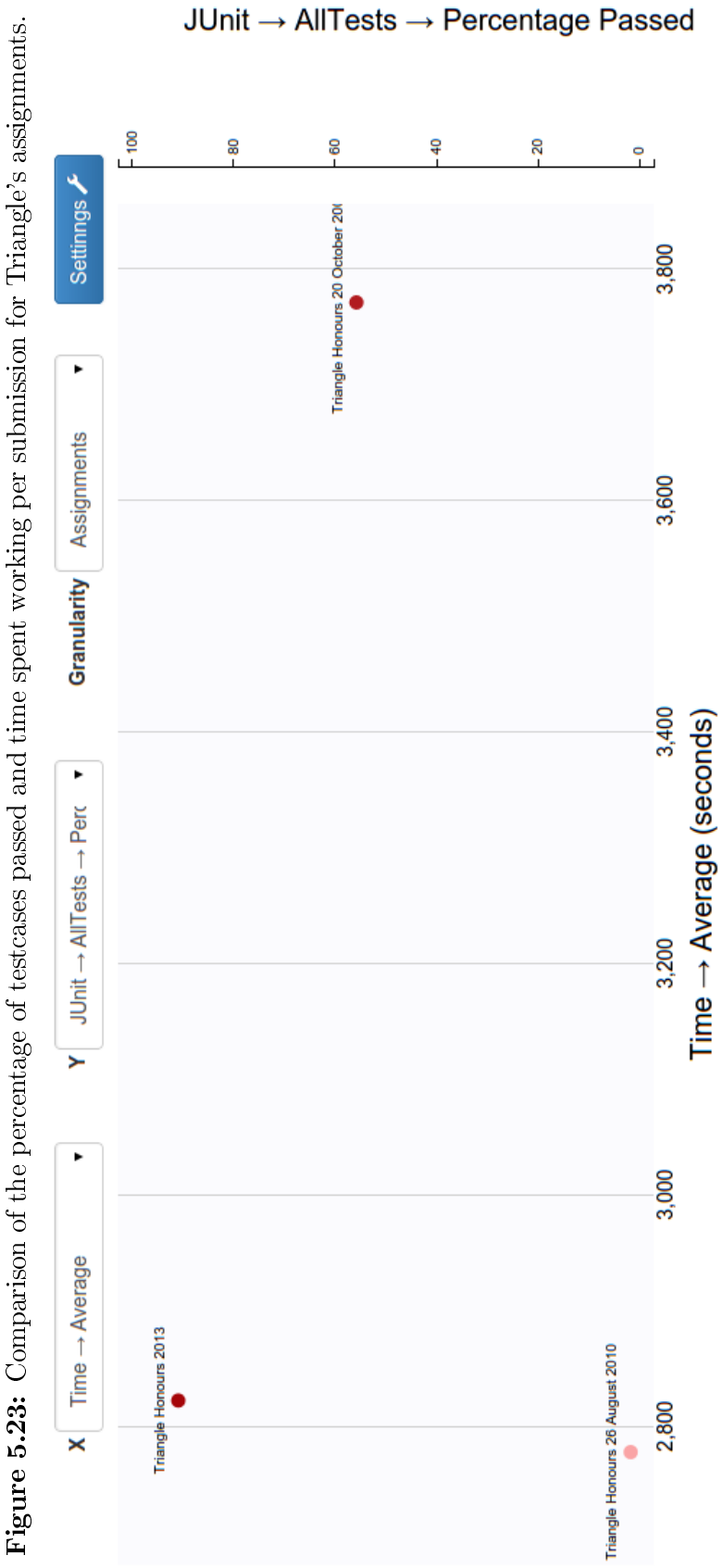


Figure 5.23: Comparison of the percentage of testcases passed and time spent working per submission for Triangle's assignments.

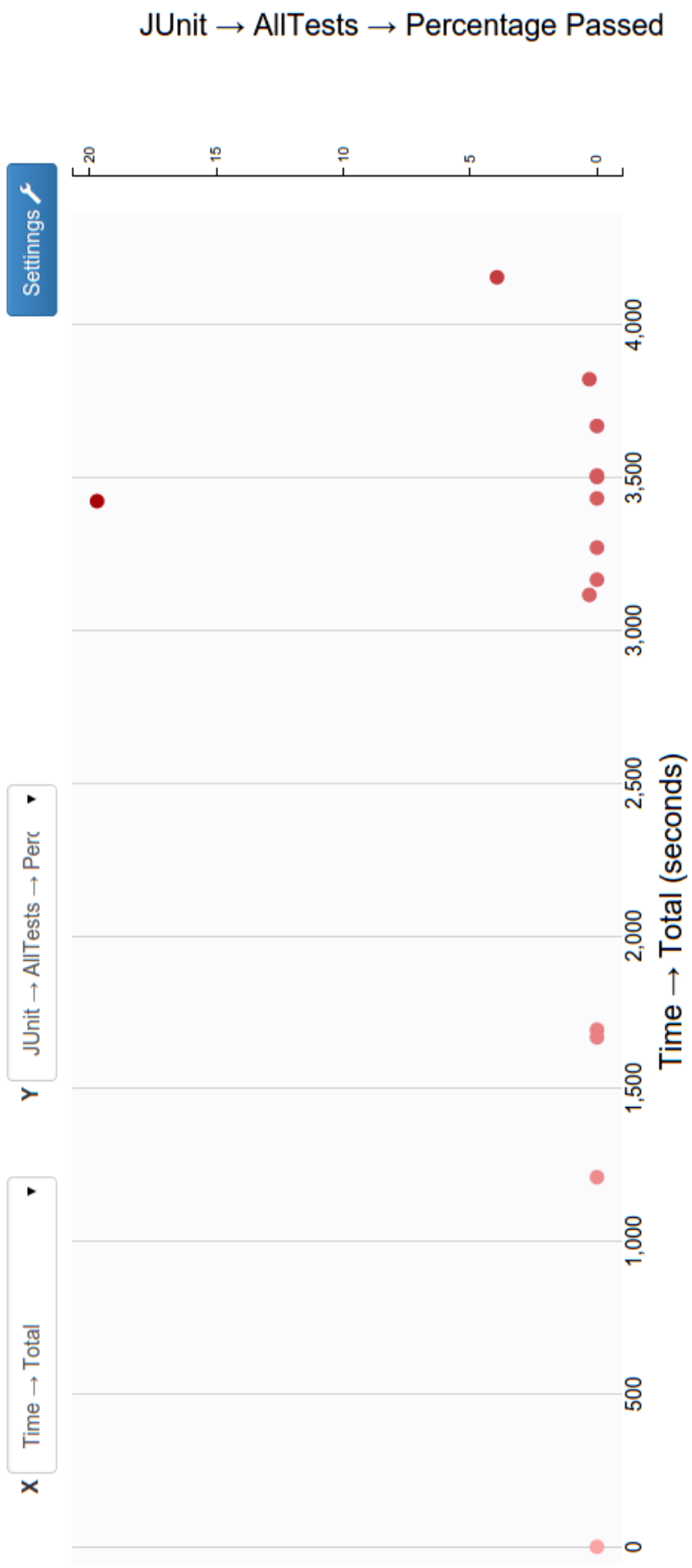


Figure 5.24: Comparison of the percentage of testcases passed and time spent working per submission for Triangle's 2010 assignment.

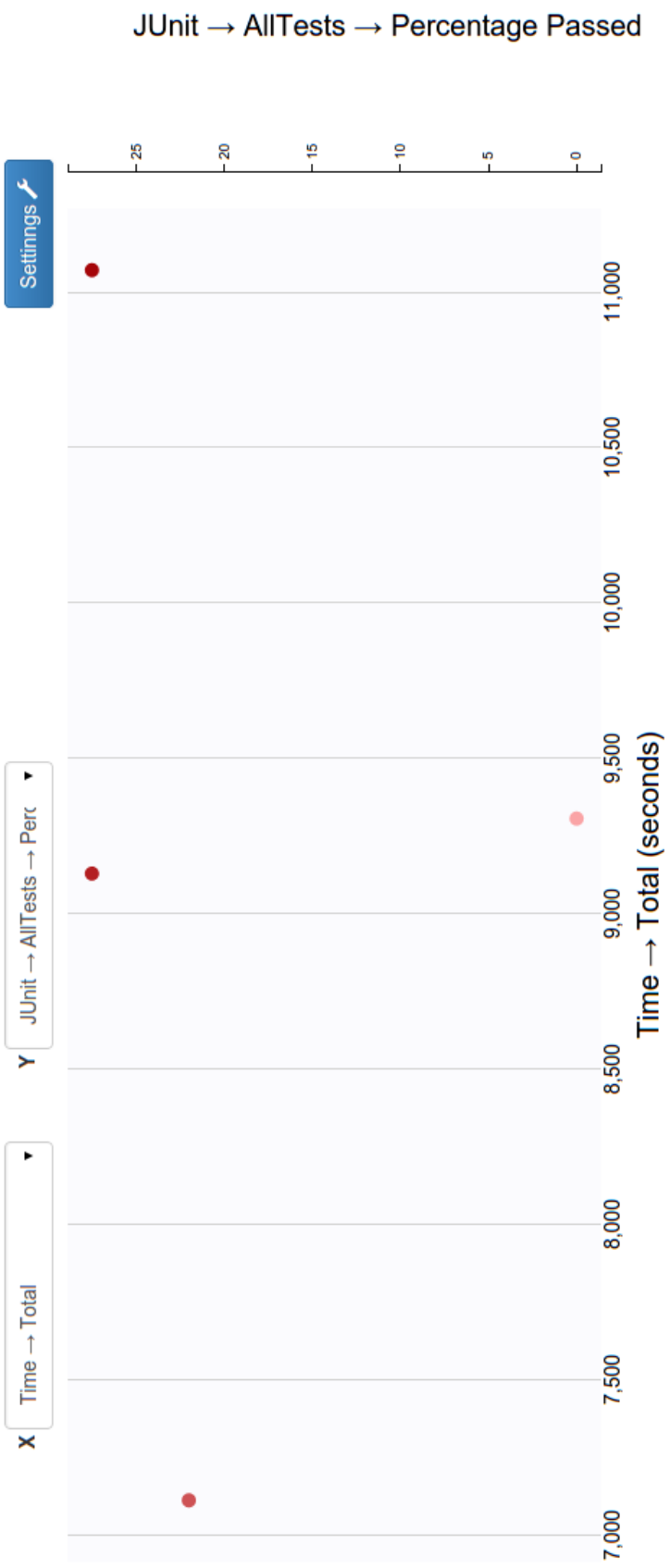


Figure 5.25: Comparison of the percentage of testcases passed and time spent working per submission for Welcome's 2010 assignment.

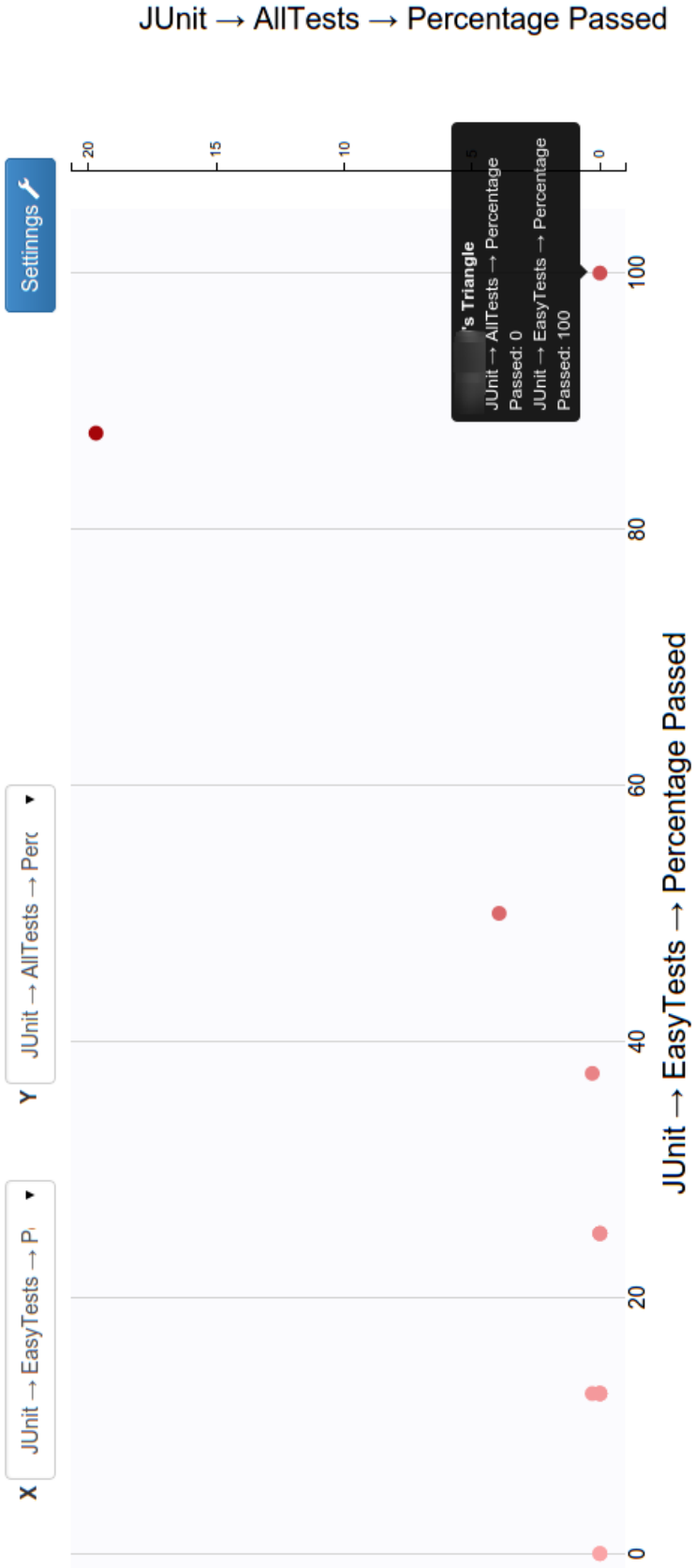


Figure 5.26: Comparison of the percentage of AllTests.java and EasyTests.java's testcases passed for Triangle's 2010 assignment.

only wait for a limited period of time instead of the original indefinite time period. Figure 5.28 clearly shows this in the form of the differences between the original source code and one of these users' final snapshot. This does remove the deadlock by automatically releasing any waiting threads, but not in the intended manner. The fact that multiple users used this method could indicate that the problem is not explained clearly enough and insufficient restrictions are placed on the users with regards to how they can solve the problem.

Another user prevented the deadlock from occurring by modifying the number of consumers used. This strengthens the argument that this problem's specification must be improved so that students know exactly what they are allowed to change. Furthermore, due to the fact that these users were so easily able to skew the results, a more stringent testing mechanism must be built for concurrency problems. This mechanism must ensure that users are only able to modify the desired parts of the program.

When we further inspect the successful submissions by looking at the differences between consecutive snapshots, we find that all bar one student constantly added and removed calls to `notifyAll()` until the bug disappeared. This seems to indicate that they had an idea of where in the code the bug occurred, but were not exactly sure how to fix it. However, this is not a surprising result since the students did not have any background in writing concurrent Java programs.

Lastly, in Figure 5.29 we can see that Findbugs is able to pinpoint the exact location of the bug as a problem area. Neither PMD nor Checkstyle complained about this use of `notify()`.

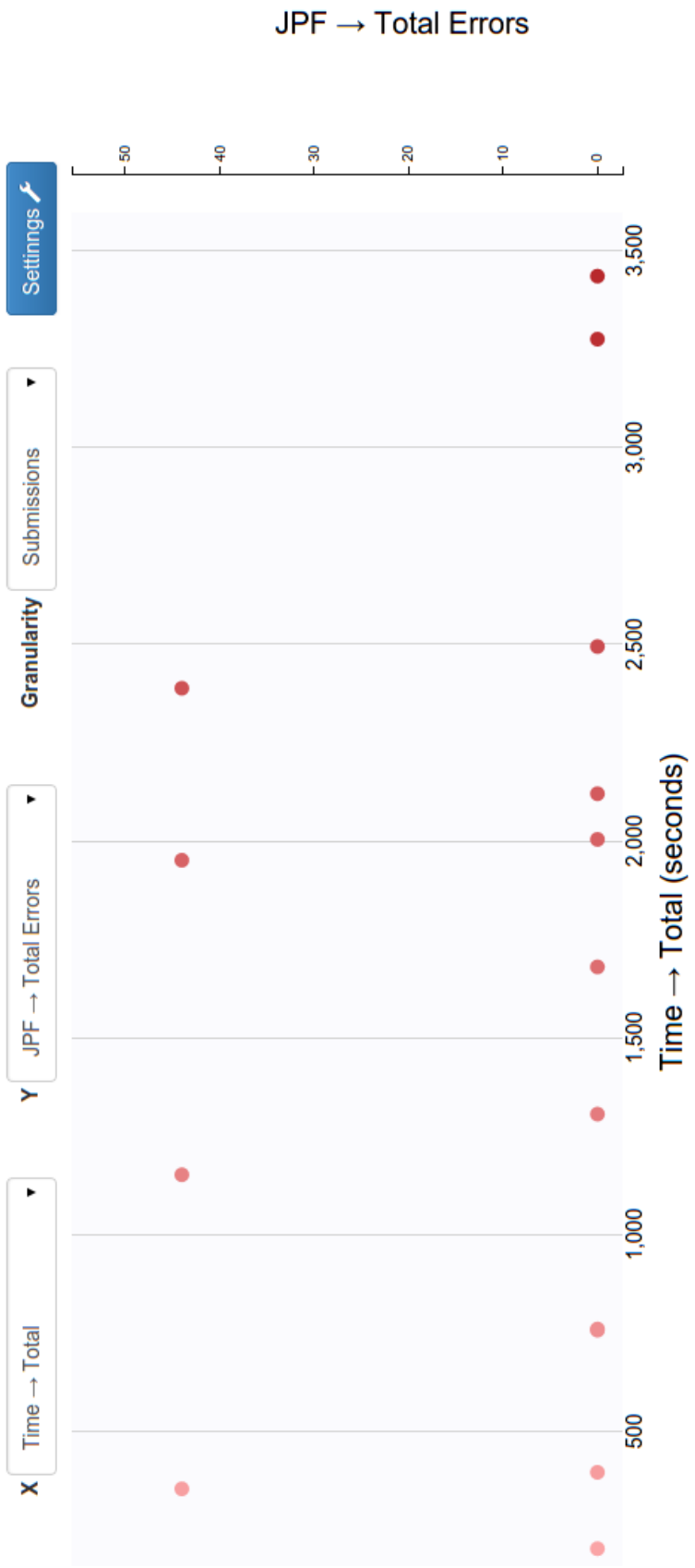


Figure 5.27: Comparison of JPF errors and time spent working per submission for BoundedBuffer's submissions.

Figure 5.28: Diff result showing the use of timed `wait()`.

```
5     public synchronized void put(Object o) throws InterruptedException {
6       while (count == size) {
7   -     wait();
8   +     wait(1000);
9       }
10      buf[in] = o;
11      ++count;
12      in = (in + 1) % size;
13  +   System.out.println("Item "+o.toString()+" put in buffer");
14      notify();
15    }
16
17    public synchronized Object get() throws InterruptedException {
18      while (count == 0) {
19  -     wait();
20  +     wait(1000);
```

Figure 5.29: Findbugs result highlighting the BoundedBuffer bug .

Bugs

Using notify() rather than notifyAll()

Priority	3
Category	Multithreaded correctness
Rank	17
Description	Using notify rather than notifyAll in boundedbuffer.BoundedBuffer.get()
Source Line	At BoundedBuffer.java:[line 91]

Using notify() rather than notifyAll()

Chapter 6

Conclusion

We have created a platform for analysing student code which provides much of the functionality available in traditional automated submission and grading systems. What makes our system unique is its configurable toolchain and use of fine-grained submissions. Furthermore, the analysis and visualisation features provided by Impendulo significantly expand the insight there is to be gained from each tool in the toolchain.

Normally, software analysis tools take time to understand, setup and configure. Impendulo significantly lowers this overhead by providing a number of integrated tools in its default toolchain and allowing users to easily integrate any number of other tools into the system. Once a user adds a tool, it can repeatedly be used without them having to do any additional work.

Furthermore, not all tools provide their results in a user-friendly format, and those that do, do not allow one to easily compare different results. Impendulo addresses this issue by parsing a tool's results and providing the user with multiple ways to view the data such as reports, tables and charts. These analysis and visualisation features of Impendulo significantly expand the insight there is to be gained from each tool by allowing data to be compared and viewed at different granularities.

In chapter 5 we showed that these features are feasible by allowing us to identify issues that users experienced when working on assignments. Furthermore, we showed that it has the potential for identifying underlying issues in larger groups. However, due to the small sample sizes and the fact that the experiments did not follow a stringent method means that this is only a conjecture.

6.1 Future Work

There are several ways in which Impendulo can be improved upon.

Firstly, adding new tools currently requires the Impendulo installation to be recompiled and restarted. It would therefore be a good idea to modify the

tool configuration setup in such a manner that this is no longer required.

Secondly, the interface with which the database is accessed is currently coupled too tightly to MongoDB. This should be re-implemented in a more generic manner so that any database can be used with Impendulo.

The setup in which Impendulo currently runs tools only restricts the duration which each tool can run for. This is not ideal since this could result in a tool bringing down a node by overusing its resources. In order to prevent this from happening, a secure sandbox for each tool to run in could be implemented. Ideally this would restrict the memory, CPU and storage access of the executing code. Furthermore, it should prevent or restrict access to certain vulnerable components of the operating system such as the network or the filesystem. Some of the potential solutions to this issue are: chroot [39], Docker [40], SELinux [41] or Linux Containers [42].

In theory it should be a simple task to port Intlola's functionality to another editor. However one cannot be certain of this until it has been attempted. It would be advisable to target a less advanced editor than Eclipse so that we can determine what the minimum feature set required by Intlola is. A good candidate for this would be Vim [43] since it is commonly used when writing programs in C, which is a common language in undergraduate curricula.

Up until now Impendulo has only been used sparingly in some Computer Science Honours classes. In order to evaluate to what degree Impendulo can be used in a course, it needs to be fully integrated into a module. This will also allow us to see how usable and valuable the feedback is from a student's perspective. Preferably this will be an undergraduate module since the classes are typically large and students are still at a very early stage of their development as a programmer.

The statistical analysis capabilities of Impendulo are still very crude. This is an area which must be addressed if Impendulo is to progress to a level where it can provide true insight into the programming habits and problem areas of students.

Currently communication between Intlola and Impendulo is conducted over a plain TCP connection and all data is sent unencrypted. It would therefore be prudent to secure this communication by using a cryptographic protocol such as Transport Layer Security (TLS) [44].

Communication between the different components which comprise Impendulo is currently limited to being conducted over RabbitMQ. Allowing this medium to be chosen according to the needs of the installation would be a great boon. The inter-process communication interface would therefore need to be modified so that it uses a generic interface instead. This would allow any communication protocol which implements the interface to be used by Impendulo.

Appendices

Appendix A

Tool Integration

In order for a new tool to be added to Impendulo's toolchain, it is required to implement a number interfaces. This appendix describes how to implement each of these.

A.1 tool.T

This interface is used when the tool is run. It primarily serves as a way to execute the tool and retrieve the results thereof. In Listing A.1 we can see how the interface is specified.

Listing A.1: The tool.T interface

```
1  T interface {
2    //Name retrieves the Tool's name.
3    Name() string
4    //Lang retrieves the language which the Tool is
       used for.
5    Lang() project.Language
6    //Run runs the tool on a given file.
7    Run(fileId bson.ObjectId, target *Target) (result
       .Tooler, error)
8  }
```

The `Name` and `Lang` functions are straightforward and can be implemented as one-liners. `project.Language` is just an alias for `string` and represents a programming language such as "Java" or "C". However, the `Run` method is a bit more complicated. It's first parameter is the database ID of the file on which the tool is going to be run. The second parameter tells the tool where the file is stored. As can be seen in Listing A.2, it basically provides an easy way to access the different components of a file's path.

Listing A.2: The tool.Target struct

```
1 //Target stores information about a target file.
```

```
2  Target struct {
3    //Name of the file. Example: ~/dir/package/File.
       java -> File
4    Name string
5    //The file's package. Example: ~/dir/package/File.
       java -> package
6    Package string
7    //The file's extension. Example: ~/dir/package/File
       .java -> java
8    Ext string
9    //The file's directory (not including the package).
        Example: ~/dir/package/File.java -> ~/dir
10   Dir  string
11   Lang project.Language
12 }
```

The tool should then use the `target` to run on the file. In Listing A.3 we can see `RunCommand`, a helper function which can be used to run tools. The first element in the `args` array is the command that the function will invoke; the rest of the elements are arguments it will pass to the command. `stdin` can be used to send data to the commands standard input. If it is not required, `nil` (go's version of `null`) can simply be passed to the function. `max` is the maximum duration which the tool is allowed to run for.

Listing A.3: The function used to run external commands

```
1 func RunCommand(args []string, stdin io.Reader, max
     time.Duration) (*Result, error)
```

After executing the command, the function returns the result thereof in a `tool.Result` struct (Listing A.4). This contains the data from the command's standard output and error.

Listing A.4: The tool.Result struct

```
1   Result struct {
2     StdOut, StdErr []byte
3   }
```

Therefore, after using
`RunCommand` to execute the tool, the tool has the resulting raw data. According to the `Run` method's signature, it must return a
`result.Tooler`. The tool must therefore parse the raw data and store it in a data structure which implements this interface.

A.2 result.Tooler

In Listing A.5 we can see that most of the methods it requires are straightforward getters. The `OnGridFS` method depends on the size of the result. If a result is larger than 16 megabytes it is partially stored in MongoDB's grid filesystem. The first line of the interface simply says "R". This means that if one wishes to implement `result.Tooler`, they must also implement `result.R`.

Listing A.5: The result.Tooler interface

```
1  //Tooler is used to store a tool's result data.
2  Tooler interface {
3    R
4    //GetId retrieves the result's db id.
5    GetId() bson.ObjectId
6    //GetTestId retrieves the result's test db id (if
         applicable).
7    GetTestId() bson.ObjectId
8    //OnGridFS returns true if this result is
       partially stored on GridFS, false otherwise.
9    OnGridFS() bool
10   //SetReport sets this result's tool report. Used
       mainly to move data from/to GridFS
11   SetReport(Reporter)
12 }
```

In Listing A.6 we see that the `result.R` interface also only requires the implementation of getters. Here `Reporter()` returns the actual analysis data of the result.

Listing A.6: The result.R interface

```
1  R interface {
2    //GetName retrieves the specific name associated
       with this result.
3    GetName() string
4    //GetType retrieves the name of this report's
       tool.
5    GetType() string
6    //Retrieves the report generated by the
       associated tool stored in this result.
7    Reporter() Reporter
8    //Retrieves the file associated with the result's
       db id.
9    GetFileId() bson.ObjectId
10 }
```

If all of this has been successfully implemented, the tool can be run by Impendulo and its results stored.

A.3 result.Displayer

If a tool's results are to be displayed by Impendulo, they must first implement the `result.Displayer` interface. In Listing A.7 we can see that there is only one further method to implement, `Template`.

Listing A.7: The result.Displayer interface

```
1  //Displayer is used to display result reports.
2  Displayer interface {
3    R
4    //Template retrieves the name of a HTML template.
5    Template() string
6  }
```

This method just provides the path to an HTML template file which specifies how the result should be displayed. The file must use go's templating engine. All template commands are enclosed by double braces: `{{ template commands }}`. The file is defined to be part of the "result" template. This simply means that the file must start with a `{{define "result"}}` tag and end with a `{{end}}` tag. The result's analysis data (retrieved by a call to `Reporter`) can be accessed by using ".Report" in the template. In Listing A.8 we can see how an HTML template file can be used in conjunction with JavaScript (Listing A.9) to render a tool's analysis results (Listing A.10).

Listing A.8: Checkstyle's HTML template

```
1  {{define "result"}}
2  {{$report := .Report}}
3  {{if $report.Success}}
4  <h4 class="text-success">No problems detected.</h4>
5  {{else}}
6  {{$fileName := .ctx.Browse.File}}
7  {{$file := $report.File $fileName}}
8  {{$rid := $report.Id.Hex}}
9  <h4 class="text-danger">{{$report.Errors}} problems
     detected.</h4>
10 <div class="panel-group" id="checkstyleaccordion">
11 </div>
12 {{range $file.Errors}}
13 {{$errorName := shortName .Source}}
14 {{$package := package $errorName}}
15 {{$class := class $errorName}}
```

```
{{$msg := .Message}}
<script type="text/javascript" language="javascript"
   src="/static/js/checkstyle.js"></script>
<script>
var eid = "error{{.Id.Hex}}";
CheckstyleResult.addFile("checkstyleaccordion", eid,
   "{{$package}}", "{{$class}}", "{{.Severity}}", "{{
   $msg}}");
</script>
{{range .Lines}}
<script>
CheckstyleResult.addLine("{{$rid}}", eid, "{{.}}", "
   {{$errorName}}", "{{$msg}}");
</script>
{{end}}
{{end}}
{{end}}
{{end}}
```

Listing A.9: Checkstyle's JavaScript file

```
var CheckstyleResult = {
  addFile: function(parentId, eid, pkg, clss,
     severity, message, lines, resultID) {
    var pkgID = parentId + pkg;
    var aID = 'accordion' + pkgID;
    if ($('#' + aID).length == 0) {
      $('#' + parentId).append('<div class="panel
         panel-default"><div class="panel-heading"><a
         class="accordion-toggle" data-toggle="
         collapse" data-parent="' + parentId + '" href
         ="#' + pkgID + '"><h4 class="text-center">' +
          pkg + '</h4></a></div><div id="' + pkgID + '
         " class="panel-collapse collapse"><div class
         ="accordion-inner"><div class="panel-group"
         id="' + aID + '"></div></div></div></div>');
    }
    $('#' + aID).append('<div class="panel panel-
       default"><div class="panel-heading"><a class="
       accordion-toggle" data-toggle="collapse" data-
       parent="' + aID + '" href="#' + eid + '"><h5>'
       + clss + '</h5></a></div><div id="' + eid + '"
       class="panel-collapse collapse"><div class="
       accordion-inner"><dl class="dl-horizontal"><dt>
       Lines</dt><dd class="lines"></dd><dt>Severity</
```

```
        dt><dd>' + severity + '</dd><dt>Description</dt
        ><dd>' + message + '</dd></dl></div></div></div
        >');
  },
  addLine: function(resultID, eid, num, title,
     message) {
    var lid = eid + num;
    $('#' + eid + ' dd.lines').append('<a href="#" id
       ="' + lid + '"> ' + num + '; </a>');
    var info = {
      'title': title,
      'content': message
    };
    AnalysisView.addCodeModal(lid, resultID, info,
       num, num);
  }
}
```

Listing A.10: Checkstyle's analysis report

```
//Report represents the result of running
   Checkstyle on a Java source file.
Report struct {
  Id       bson.ObjectId
  Version  string `xml:"version,attr"`
  Errors   int
  Files    []*File `xml:"file"`
}
//File represents a file on which checkstyle was
   run and all errors found in it.
File struct {
  Name   string `xml:"name,attr"`
  Errors Errors `xml:"error"`
}
Errors []*Error
//Error represents an occurrence of an error
   detected by checkstyle.
Error struct {
  Id       bson.ObjectId
  Line     int            `xml:"line,attr"`
  Column   int            `xml:"column,attr"`
  Severity string         `xml:"severity,attr"`
  Message  template.HTML `xml:"message,attr"`
  Source   string         `xml:"source,attr"`
  Lines    []int
```

```
23    }
```

A.4 result.Valuer

In order for a result to be shown in Impendulo's tables and plotted in its charts, it must implement the `result.Valuer` interface which can be seen in Listing A.11.

Listing A.11: The result.Valuer interface

```
1   //Valuer is used to provide values which can be
       used in charts and tables.
2   Valuer interface {
3     R
4     //Values retrieves all of the result's values.
5     Values() []*Value
6     //Value retrieves a single value with the same
        name as n.
7     Value(n string) (*Value, error)
8     //Types retrieves the names of the result's
        values.
9     Types() []string
10  }
11  Value struct {
12    Name    string
13    V       float64
14    FileId bson.ObjectId
15  }
```

Appendix B

Projects

Here follows descriptions of the various projects used in the experiments conducted with Impendulo.

B.1 Triangle

We are given a triangle of numbers such as the one pictured beneath. By starting at the top of the triangle and moving to adjacent numbers on the row below until we reach the last row, we can trace a *path* from top to bottom. The *weight* of a path is the sum of all the numbers we have visited on the path.

$$\begin{array}{ccccccc} & & & \underline{3} & & & \\ & & \underline{7} & & 4 & & \\ & 2 & & \underline{4} & & 6 & \\ 8 & & 5 & & \underline{9} & & 3 \end{array}$$

The goal is to find the maximum weight of any path in a given triangle. In this example, there is only one path that delivers the maximum weight. The numbers of this path has been underlined, and the weight is $3+7+4+9=23$

Your task is to write a function in the `triangle.Triangle` class that accepts one parameter: an array `tri` that represents the numbers of the triangle. Your routine must return the weight of a heaviest path as an integer. Note that there may be more than one path that have this weight.

```
public int maxpath(int tri[][])
```

The first array element `tri[0]` is a one-dimensional array that contains the top number of the triangle (`tri[0][0]`). The second element `tri[1]` is a two dimensional array that contains the two numbers in the second row of the triangle (`tri[1][0]` and `tri[1][1]`), and so on.

In other words, the example triangle above is passed to your routine as:

```
{ { 3 },
  { 7, 4 },
  { 2, 4, 6 },
  { 8, 5, 9, 3 } }
```

You may assume that:

- The triangle contains at least one and at most 100 rows.
- Each number x in the triangle satisfies $0 \leq x < 10^6$.
- The answer will therefore be less than 10^8.

B.2 KSelect

Given two pairs of integers (a_1, a_2) and (b_1, b_2) we can compare them by first comparing the first component and then the second. For example,

$$(a_1, a_2) < (b_1, b_2) \quad \text{if and only if} \quad a_1 < b_1 \quad \text{or} \quad a_1 = b_1 \wedge a_2 < b_2.$$

The *k-select problem* is to find the k-th smallest pair in a list of pairs. When $k < 0$, the task is to find the $-k$-th largest pair. If $k = 0$, or if the absolute value of k is greater than the length of the list, we shall say that the answer is zero. For example, given the list

$$\underset{1}{(3,1)} \quad \underset{2}{(4,1)} \quad \underset{3}{(5,9)} \quad \underset{4}{(2,6)} \quad \underset{5}{(5,3)} \quad \underset{6}{(5,8)}$$

then

$$(2,6) \;<\; (3,1) \;<\; (4,1) \;<\; (5,3) \;<\; (5,8) \;<\; (5,9)$$

and we know that:

$$\begin{aligned} 1^{\text{st}} &\text{ smallest pair } (2,6) \text{ is in position } 4, \\ 4^{\text{th}} &\text{ smallest pair } (5,3) \text{ is in position } 5, \\ -1^{\text{th}} &\text{ smallest pair } (5,9) \text{ is in position } 3 \text{ and,} \\ -4^{\text{th}} &\text{ smallest pair } (4,1) \text{ is in position } 2. \end{aligned}$$

Your task is to write a routine in the `kselect.KSelect` class that accepts two parameters: (1) the value of k and (2) the list of integer pairs, stored in a single array. Your routine must return the position of the k-smallest pair as an integer.

```
public int kselect(int k, int values[])
```

For example,

`kselect( 1, {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8})` should return `4`
`kselect(-3, {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8})` should return `5`
`kselect( 7, {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8})` should return `0`

You may assume that

- the list contains at least one and at most 10,000 pairs,
- none of the pairs are equal, and
- each integer x in the list satisfies $0 \leq x \leq 10^6$.

B.3 BoundedBuffer

This program is an example of the classic Producer/Consumer concurrency problem. The crux of it is that there are two classes of process, a producer and a consumer, which share a common, fixed-size buffer as a queue. Each producer continuously generates data and puts it into the buffer. Simultaneously, each consumer consumes the data one piece at a time. However we must ensure that producers don't add data into a full buffer and consumers don't try to remove data from an empty buffer.

There is a concurrency bug in this program. Your task is to identify and fix it.

The code for this problem can be seen in Listing C.3.

B.4 oldclassic

This problem is based on a defect found in Remote Agent (RA), an AI-based spacecraft controller which flew on board of *Deep Space 1*. This resulted in RA experiencing a deadlock in real flight.

The program consists of an Event class and two Thread classes. Event has a local counter variable and two synchronized methods. `wait_for_event` is used to wait on the event and `signal_event` signals the event, thereby releasing all waiting threads. The counter is incremented (modulo the number of threads) whenever `signal_event` is called, so that events can be caught while tasks are executing. A task must therefore only call `wait_for_event` if this counter has not changed (since this implies that no events were missed).

Each Thread class has two Events, one which it uses to signal the other thread and the other which it receives signals on.

Your task is to identify and fix the bug in this program.

The code for this problem can be seen in Listing C.2.

B.5 Problematic

`Problematic.java` is a concurrent program which contains a bug due to a race condition. It is up to you to identify and fix it.

The code for this problem can be seen in Listing C.4.

B.6 TriType

Suppose we are given three integers that represent the sides of a triangle. In some cases, the integer may not form a proper triangle (for example if one of the sides is -1). If they do form a valid triangle, it is called *equilateral* if all three sides are equal, it is called *isosceles* if exactly two sides are equal, and otherwise it is classified as *scalene*.

Your task is to complete the `classify()` function in the `tritype.TriType` class that accepts three Java `int` parameters which represent the triangle sides. The routine must also return an `int` that represents the classification of the triangle:

1	equilateral
2	isosceles
3	scalene
4	invalid triangle

You may assume that:

- The parameters take values from `Integer.MIN_VALUE` to `Integer.MAX_VALUE`, inclusive.
- The parameters are valid integers and there is no need to test for values such as "`abc`".
- All triangles can be classified as exactly one of the four types.

B.7 Watersheds

Geologists sometimes divide an area of land into different regions based on where rainfall flows down to. These regions are called *drainage basins*. Given an elevation map (a 2-dimensional array of altitudes), label the map such that locations in the same drainage basin have the same label, subject to the following rules.

- From each cell, water flows down to at most one of its 4 neighbouring cells.

- For each cell, if none of its 4 neighbouring cells has a lower altitude than the current cell's, then the water does not flow, and the current cell is called a sink.
- Otherwise, water flows from the current cell to the neighbour with the lowest altitude.
- In case of a tie, water will choose the first direction with the lowest altitude from this list: North, West, East, South.

Every cell that drains directly or indirectly to the same sink is part of the same drainage basin. Each basin is labeled by a unique lower-case letter, in such a way that, when the rows of the map are concatenated from top to bottom, the resulting string is lexicographically smallest. This means that the basin of the most North-Western cell is always labeled `a` and, as we scan the basins left to right, top to bottom, the next basin we encounter is `b`, then `c`, then `d`, and so on.

Example 1
In this example, the upper-right and lower-left corners are sinks. Water from the diagonal flows towards the lower-left because of the lower altitude (5 versus 6).

```
9 6 3                a b b          ( ↓ → •
5 9 6      produces  a a b            ↓ ← ↑
3 5 9                a a a            • ← ← )
```

Example 2

```
1 2 3 4 5                  a a a a a
2 9 3 9 6                  a a b b a
3 3 0 8 7      produces    a b b b a
4 9 8 9 8                  a b b b a
5 6 7 8 9                  a a a a a
```

Example 3

```
8 8 8 8 8 8 8 8                  a b c d e f g h
8 8 8 8 8 8 8 8      produces    i j k l m n o p
```

Write a routine in the `watersheds.Watersheds` class that accepts four parameters:

1. the height `h` of the map,

2. the width `w` of the map,
3. the altitudes in a two-dimensional array `alt`, and
4. a character array `basins` that your routine must fill in.

The signature of the routine is:

```
public void drainage(int h, int w, int alt[][], char basins[][])
```

The northern row of the map is `alt[0]`, and the southern row is `alt[h-1]`. The northwest (top left) altitude is `alt[0][0]`, and the southeast (bottom right) altitude is `alt[h-1][w-1]`. The `basins` array has already been allocated and is filled with `?` characters.

You may assume that

- $1 \leq$ `h` ≤ 100 and $1 \leq$ `w` ≤ 100,
- $0 \leq$ altitudes $< 10{,}000$, and
- there are at most 26 basins.

B.8 Welcome

Your task is to write a program that can take any text and print out how many times that text contains the phrase "welcome to code jam". To be more precise, given a text string, you are to determine how many times the string "welcome to code jam" appears as a sub-sequence of that string. In other words, find a sequence `s` of increasing indices into the input string `w` such that the concatenation of `w[s[0]]`, `w[s[1]]`, ..., `w[s[18]]` is the string "welcome to code jam". The result of your calculation might be huge, so for convenience we would only like you to find the last 4 digits.

Your solution should be written in a function in the `welcome.Welcome` class that has the following signature:

```
public static int count(String s)
```

You may assume that:

- The input string contains only lower-case letters and spaces.
- No line starts or ends with a space.
- Each string is no longer than 500 characters.

Appendix C

Code

Listing C.1: TriType Testcases

```
1   public class UserTests extends TestCase {
2     public void testTriType() {
3       assertEquals("Expected scalene.", 3, new
           TriType().classify(8, 9, 2));
4     }
5     @Test
6     public void testTriType2() {
7       assertEquals("Expected equilatoral.", 1, new
           TriType().classify(8, 8, 8));
8     }
9     @Test
10    public void testTriType3() {
11      assertEquals("Expected isosceles.", 2, new
           TriType().classify(3, 3, 2));
12    }
13    @Test
14    public void testTriType4() {
15      assertEquals("Expected isosceles.", 2, new
           TriType().classify(3, 2, 2));
16    }
17    @Test
18    public void testTriType5() {
19      assertEquals("Expected not a triangle.", 4, new
            TriType().classify(2, 9, 2));
20    }
21    @Test
22    public void testTriType6() {
23      assertEquals("Expected not a triangle.", 4, new
            TriType().classify(-9, 9, 2));
```

```
24      }
25      @Test
26      public void testTriType7() {
27        assertEquals("Expected not a triangle.", 4, new
              TriType().classify(2, 9, 6));
28      }
29    }
```

Listing C.2: oldclassic

```
1  package oldclassic;
2
3  public class oldclassic {
4    public static void main(String[] args) {
5      Event new_event1 = new Event();
6      Event new_event2 = new Event();
7      FirstTask task1 = new FirstTask(new_event1,
           new_event2);
8      SecondTask task2 = new SecondTask(new_event1,
           new_event2);
9      task1.start();
10     task2.start();
11   }
12 }
13
14 // ------- shared objects implemented as monitors
15 class Event {
16   int count = 0;
17
18   public synchronized void signal_event() {
19     count = (count + 1) % 3;
20     notifyAll();
21   }
22
23   public synchronized void wait_for_event() {
24     try {
25       wait();
26     } catch (InterruptedException e) {
27     }
28   }
29 }
30
31 // ------- the two concurrent threads using the
       monitors
32 class FirstTask extends Thread {
```

```
33    Event event1;
34    Event event2;
35    int count = 0;
36
37    public FirstTask(Event e1, Event e2) {
38      this.event1 = e1;
39      this.event2 = e2;
40    }
41
42    public void run() {
43      count = event1.count;
44      while (true) {
45        System.out.println("1");
46        if (count == event1.count) {
47          event1.wait_for_event();
48        }
49        count = event1.count;
50        event2.signal_event();
51      }
52    }
53  }
54
55  class SecondTask extends Thread {
56    Event event1;
57    Event event2;
58    int count = 0;
59
60    public SecondTask(Event e1, Event e2) {
61      this.event1 = e1;
62      this.event2 = e2;
63    }
64
65    public void run() {
66      count = event2.count;
67      while (true) {
68        System.out.println("  2");
69        event1.signal_event();
70        if (count == event2.count) {
71          event2.wait_for_event();
72        }
73        count = event2.count;
74      }
75    }
76  }
```

Listing C.3: BoundedBuffer

```
1  package boundedbuffer;
2
3  public class BoundedBuffer {
4
5    static int BUFFER_SIZE = 1;
6    static int N_PRODUCERS = 4;
7    static int N_CONSUMERS = 4;
8    static Object DATA = "fortytwo";
9    //--- the bounded buffer implementation
10   protected Object[] buf;
11   protected int in = 0;
12   protected int out = 0;
13   protected int count = 0;
14   protected int size;
15
16   public BoundedBuffer(int size) {
17     this.size = size;
18     buf = new Object[size];
19   }
20
21   public synchronized void put(Object o) throws
        InterruptedException {
22     while (count == size) {
23       wait();
24     }
25     buf[in] = o;
26     ++count;
27     in = (in + 1) % size;
28     notify();
29   }
30
31   public synchronized Object get() throws
        InterruptedException {
32     while (count == 0) {
33       wait();
34     }
35     Object o = buf[out];
36     buf[out] = null;
37     --count;
38     out = (out + 1) % size;
39     notify();
40     return (o);
```

```
41    }
42
43    //--- the producer
44    static class Producer extends Thread {
45      static int nProducers = 1;
46      BoundedBuffer buf;
47
48      Producer(BoundedBuffer b) {
49        buf = b;
50        setName("P" + nProducers++);
51      }
52
53      public void run() {
54        try {
55          while(true) {
56            buf.put(DATA);
57          }
58        } catch (InterruptedException e){}
59      }
60    }
61
62    //--- the consumer
63    static class Consumer extends Thread {
64      static int nConsumers = 1;
65      BoundedBuffer buf;
66
67      Consumer(BoundedBuffer b) {
68        buf = b;
69        setName( "C" + nConsumers++);
70      }
71
72      public void run() {
73        try {
74          while(true) {
75              Object tmp = buf.get();
76          }
77        } catch(InterruptedException e ){}
78      }
79    }
80
81    //--- the test driver
82    public static void main(String [] args) {
83      readArguments( args);
84      BoundedBuffer buf = new BoundedBuffer(BUFFER_SIZE
```

```
        );
85    for (int i=0; i<N_PRODUCERS; i++) {
86      new Producer(buf).start();
87    }
88    for (int i=0; i<N_CONSUMERS; i++) {
89      new Consumer(buf).start();
90    }
91  }
92
93  static void readArguments (String[] args){
94    if (args.length > 0){
95      BUFFER_SIZE = Integer.parseInt(args[0]);
96    }
97    if (args.length > 1){
98      N_PRODUCERS = Integer.parseInt(args[1]);
99    }
100   if (args.length > 2){
101     N_CONSUMERS = Integer.parseInt(args[2]);
102   }
103 }
104 }
```

Listing C.4: Problematic

```
1  package problematic;
2  public class Problematic {
3
4    static class Authentication {
5      static int counter = 0;
6      static synchronized void verify() {
7        // Do a little work
8        try { Thread.sleep(10); } catch (
             InterruptedException x) { }
9      }
10     static synchronized boolean update() {
11       return counter++ < 100;
12     }
13   }
14
15   static class ClientAccountView {
16     static int counter = 0, backupCounter = 0;
17     static synchronized void updateAccount() {
18       ++counter;
19       Authentication.verify();
20       ++backupCounter;
```

```
21      }
22    }
23
24    static class ManagerAccountView extends
         ClientAccountView {
25      static synchronized void check() {
26        if (counter != backupCounter) {
27          System.out.println("BUG!");
28          System.exit(1);
29        }
30      }
31    }
32
33    static class Client extends Thread {
34      public void run() {
35        ClientAccountView.updateAccount();
36      }
37    }
38
39    static class Manager extends Thread {
40      public void run() {
41        ManagerAccountView.check();
42      }
43    }
44
45    static class AuthenticationServer extends Thread {
46      public void run() {
47        while (Authentication.update()) {
48          // Do some work
49          try { Thread.sleep(100); } catch (
               InterruptedException x) { }
50        }
51      }
52    }
53
54    public static void main(String[] args) {
55      new Client().start();
56      new Manager().start();
57    }
58  }
```

Appendix D

Intlola API

The Intlola API is used to facilitate communication between Intlola and Impendulo. The Intlola API is a communication protocol which makes use of JSON to send data and instructions to Impendulo. Any advanced responses Impendulo is required to send are sent as JSON. When Impendulo is merely required to acknowledge that an operation was successful, it sends "ok" in plain text. We refer to this as the *OK response*. The operations defined by the API are:

Login This is used to log an existing user in to Impendulo. The required response to this request is a list of available assignments as well as any active submissions the user may have in them.

```
{''request'': ''login'',
  ''user'': ''username'',
  ''password'': ''password'',
  ''mode'': ''archive_remote or file_remote''
}
```

Register This command is used to register a new user with Impendulo and log them in to the system. The required response to this request is a list of available assignments.

```
{''request'': ''register'',
  ''user'': ''username'',
  ''password'': ''password'',
  ''mode'': ''archive_remote or file_remote''
}
```

New Submission This command is invoked when a user wishes to create and work on a new submission in an assignment. This request requires the OK response.

```
{''request'': ''submission_new'',
  ''assignmentid'': ''the id of the assignment
     in which to create
  the submission'',
  ''projectid'': ''the id of the project to
     which the assignment belongs'',
  ''time'': ''the time at which the assignment
     was started''
}
```

Continue Submission When a user wishes to continue working on one of their own existing submissions, this command is used. This requires the OK response.

```
{''request'': ''submission_continue'',
  ''submissionid'': ''the id of the existing
     submission'',
}
```

Send The `Send` command is used to send snapshot metadata to Impendulo and to indicate that a snapshot's contents will follow it. The format of the command depends on the type of snapshot. Firstly we have the archive file format:

```
{''request'': ''send'',
  ''type'': ''archive''
}
```

Next we have the format for all other files:

```
{''request'': ''send'',
  ''type'': ''launch, test or src'',
  ''name'': ''the file's name'',
  ''package'': ''the file's package'',
  ''time'', ''the time at which the file was
     recorded.''
}
```

Impendulo responds with the OK response. Hereafter Intlola sends the contents of the file. If this is successful, the OK response is sent once more.

Log out At the end of a user's session, the `Log out` command is used. No response is required.

```
1  {''request'': ''logout''
2  }
```

If any API operation is unsuccessful, Impendulo sends the error to Intlola as plain text and terminates the session.

List of References

[1] McConnell, S.C.: *Code Complete*. 2nd edn. Microsoft Press, 2004.

[2] Jones, C.: Software Quality Metrics: Three Harmful Metrics and Two Helpful Metrics. Tech. Rep., Project Performance International (PPI), June 2012.

[3] RTI: The Economic Impacts of Inadequate Infrastructure for Software Testing. Tech. Rep., RTI International, May 2002.

[4] Gdp (current us$). 2014.
Available at: `http://data.worldbank.org/indicator/NY.GDP.MKTP.CD`

[5] McDonald, M., Musson, R. and Smith, R.: *The Practical Guide to Defect Prevention*. 1st edn. Microsoft Press, 2008.

[6] Huizinga, D. and Kolawa, A.: *Automated Defect Prevention: Best Practices in Software Management*. 1st edn. John Wiley & Sons, Inc., 2007.

[7] Visser, W. and Geldenhuys, J.: Impendulo: Debugging the Programmer. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 351–352. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0116-9.
Available at: `http://doi.acm.org/10.1145/1858996.1859071`

[8] Geldenhuys, J. and Visser, W.: Static Analysis for Bug Finding; are we going in the wrong direction? Tech. Rep., Stellenbosch University, 2010.

[9] Ihantola, P., Ahoniemi, T., Karavirta, V. and Seppälä, O.: Review of Recent Systems for Automatic Assessment of Programming Assignments. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pp. 86–93. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0520-4.
Available at: `http://doi.acm.org/10.1145/1930464.1930480`

[10] Ala-Mutka, K.M.: A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, vol. 15, no. 2, pp. 83–102, 2005. `http://dx.doi.org/10.1080/08993400500150747`.
Available at: `http://dx.doi.org/10.1080/08993400500150747`

[11] Hollingsworth, J.: Automatic Graders for Programming Classes. *Commun. ACM*, vol. 3, no. 10, pp. 528–529, October 1960. ISSN 0001-0782.
Available at: `http://doi.acm.org/10.1145/367415.367422`

[12] Isaacson, P.C. and Scott, T.A.: Automating the Execution of Student Programs. *SIGCSE Bull.*, vol. 21, no. 2, pp. 15–22, June 1989. ISSN 0097-8418.
Available at: `http://doi.acm.org/10.1145/65738.65741`

[13] Zeller, A.: Making Students Read and Review Code. *SIGCSE Bull.*, vol. 32, no. 3, pp. 89–92, July 2000. ISSN 0097-8418.
Available at: `http://doi.acm.org/10.1145/353519.343090`

[14] Joy, M., Griffiths, N. and Boyatt, R.: The BOSS Online Submission and Assessment System. *ACM Journal on Educational Resources in Computing*, vol. 5, no. 3, pp. 1–28, September 2005.
Available at: `http://eprints.dcs.warwick.ac.uk/3/`

[15] Ellsworth, C.C., Fenwick, Jr., J.B. and Kurtz, B.L.: The Quiver System. *SIGCSE Bull.*, vol. 36, no. 1, pp. 205–209, March 2004. ISSN 0097-8418.
Available at: `http://doi.acm.org/10.1145/1028174.971374`

[16] Jackson, D. and Usher, M.: Grading Student Programs Using ASSYST. *SIGCSE Bull.*, vol. 29, no. 1, pp. 335–339, March 1997. ISSN 0097-8418.
Available at: `http://doi.acm.org/10.1145/268085.268210`

[17] McCabe, T.J.: A Complexity Measure. *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, July 1976. ISSN 0098-5589.
Available at: `http://dx.doi.org/10.1109/TSE.1976.233837`

[18] Berry, R.E. and Meekings, B.A.: A Style Analysis of C Programs. *Commun. ACM*, vol. 28, no. 1, pp. 80–88, January 1985. ISSN 0001-0782.
Available at: `http://doi.acm.org/10.1145/2465.2469`

[19] English, J. and Siviter, P.: Experience with an Automatically Assessed Course. *SIGCSE Bull.*, vol. 32, no. 3, pp. 168–171, July 2000. ISSN 0097-8418.
Available at: `http://doi.acm.org/10.1145/353519.343161`

[20] Edwards, S.H.: Using Software Testing to Move Students from Trial-and-error to Reflection-in-action. *SIGCSE Bull.*, vol. 36, no. 1, pp. 26–30, March 2004. ISSN 0097-8418.
Available at: `http://doi.acm.org/10.1145/1028174.971312`

[21] Spacco, J., Hovemeyer, D., Pugh, W., Hollingsworth, J., Padua-Perez, N. and Emad, F.: Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In: *ITiCSE '06: Proceedings of the 11th annual conference on Innovation and technology in computer science education*. ACM Press, 2006. ISBN 1-59593-055-8.

[22] Tillmann, N., Halleux, J.D., Xie, T., Gulwani, S. and Bishop, J.: Teaching and Learning Programming and Software Engineering via Interactive Gaming. In: *Proc. 35th International Conference on Software Engineering (ICSE 2013), Software Engineering Education (SEE)*. May 2013.
Available at: `http://www.cs.illinois.edu/homes/taoxie/publications/icse13see-pex4fun.pdf`

[23] Eclipse Luna. December 2014.
Available at: `https://eclipse.org/`

[24] Vinoski, S.: Advanced Message Queuing Protocol. *Internet Computing, IEEE*, vol. 10, no. 6, pp. 87–89, Nov 2006. ISSN 1089-7801.

[25] RabbitMQ - what can RabbitMQ do for you? 2014.
Available at: `http://www.rabbitmq.com/features.html`

[26] MongoDB. December 2014.
Available at: `http://www.mongodb.org/`

[27] ExecutorService (Java Platform SE 7). May 2014.
Available at: `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html`

[28] The Go Programming Language. November 2014.
Available at: `https://golang.org/`

[29] Zhu, K.: Is golang goroutine a coroutine? October 2013.
Available at: `http://stackoverflow.com/questions/18058164/is-golang-goroutine-a-coroutine`

[30] The Java programming language Compiler Group. 2014.
Available at: `http://openjdk.java.net/groups/compiler/`

[31] Junit. August 2014.
Available at: `http://junit.org/`

[32] JaCoCo Java Code Coverage Library. September 2014.
Available at: `http://www.eclemma.org/jacoco/index.html`

[33] Grindstaff, C.: FindBugs, Part 1: Improve the quality of your code. May 2004.
Available at: `http://www.ibm.com/developerworks/java/library/j-findbug1/`

[34] Checkstyle. February 2014.
Available at: `http://checkstyle.sourceforge.net/`

[35] PMD. July 2014.
Available at: `http://pmd.sourceforge.net/`

[36] What is JPF? 2009.
Available at: `http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/what_is_jpf`

[37] Gnu sed. 2014.
Available at: `http://www.gnu.org/software/sed/`

[38] wc. 2013.
Available at: `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/wc.html`

[39] CHROOT(2). September 2010.
Available at: `http://man7.org/linux/man-pages/man2/chroot.2.html`

[40] Docker - Build, Ship, and Run Any App, Anywhere. December 2014.
Available at: `http://www.docker.com/`

[41] What is SELinux. December 2014.
Available at: `http://selinuxproject.org/`

[42] Linux Containers. December 2014.
Available at: `https://linuxcontainers.org/`

[43] vim online. December 2014.
Available at: `http://www.vim.org//`

[44] Dierks, T.: The Transport Layer Security (TLS) Protocol Version 1.2. August 2008.
Available at: `http://tools.ietf.org/html/rfc5246`