School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Profiling the Java Compiler for Improved Incremental Compiler Design

Philip Oliver

Supervisor: David J. Pearce

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours in Software.

Abstract

Compiling a program is a process which can take a long time, thereby breaking up a developer's workflow and productivity. Incremental compilation is a method which aims to solve this problem. Incremental compilers cache the results of previous compilations and reuse the compiled assembly or byte code of unchanged sections of a program. When designing an incremental compiler, it is vital to understand the estimated workload. Understanding which sections of the compilation pipeline find the most errors can help the designer to identify which parts require the most resources. In this project a web application has been developed to capture realistic compilation workloads using an instrumented Java compiler. Information about the stages encountered in a compilation are printed to the console from the instrumented compiler. These workloads are analysed to identify the effect on the compiler. This analysis gives insight into where in the compilation pipeline the most time is spent, during an incremental compilation session. In particular, most compilation errors occur in the parsing stage, but a large number occur in type checking for programs with use of complex type systems.

Acknowledgments

I want to acknowledge my supervisor, David J. Pearce, for his superb guidance throughout this project. Furthermore, I want to acknowledge all my lecturers and internship mentors throughout my degree. They have provided me with the knowledge and methodologies required to tackle a project like this, and I am incredibly grateful.

Contents

Figures

Chapter 1

Introduction

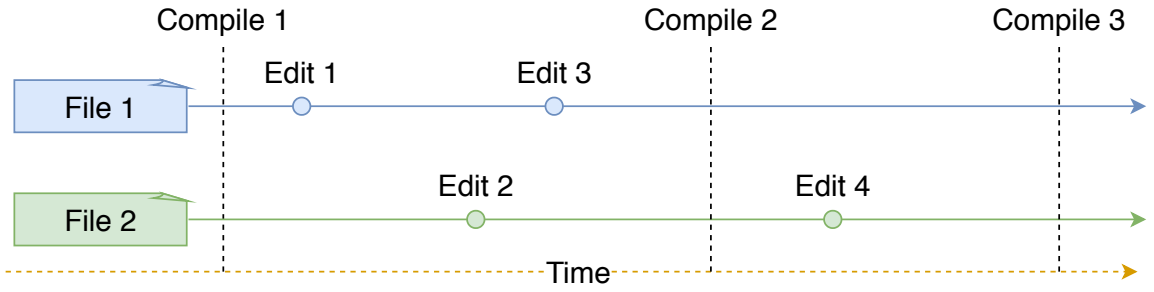
There have been a few different approaches to compilation throughout the history of software development. Initially, programs were either written in machine-readable code, such as assembly or compiled to machine-readable code by hand [1]. Grace Hopper developed the first computer-aided compiler, which laid the foundations for languages such as COBOL and Fortran [2]. After the invention of compilers, developers could write programs in higher-level languages. These higher-level languages are closer to English than the assembly code written prior and increased developer productivity [3]. Developers would now edit code and compile the software before they debug it, leading to an *edit-compile-debug* workflow. The traditional *edit-compile-debug* workflow tended to follow the approach:

1. The developer edits the code using a text editor such as Vi, Vim or Emacs.
2. The developer compiles the program using a build tool such as Make.
3. The developer then debugs the program by running it and testing it with certain inputs or testing frameworks.

This workflow lead to the invention of tools to further improve developer productivity. For example, integrated development environments (IDEs) combine all parts of the *edit-compile-debug* lifecycle. Typically, IDEs have an editor and debugging tools built-in, with most either providing their own compiler or use one present on the machine [4]. Furthermore, IDEs such as these usually provide continuous building features where recompilation occurs automatically after an edit. Continuous building means the developer continually receives feedback about mistakes in their code or sections which cannot compile. More recent developments in compiler design are distributed and web-based IDEs, such as Visual Studio Code or Cloud9 IDE, which facilitate collaborative and multi user development. These IDEs offer a web interface where developers can program and compile the code on a server. This architecture requires sending software over a network for compilation. If the software is a considerable size, this communication between the client and server will take some time, depending on factors such as connection latency and bandwidth. With increasingly more extensive programs and compiler workloads growing due to methods such as continuous building, finding a new approach to software compilation is essential.

1.1 The Compilation Problem

When a developer is working on a program, often they are only changing a small section of the software, and then recompiling and debugging [5]. Batch compilers are relatively unintelligent compilers, and recompile the entire program after each change. Compiling the entire program to debug a few small changes is resource-intensive and inefficient. The below shows an example of how a typical batch compilation session may occur.



At `Compile 1` both files need compiling. However, between `Compile 2` and `Compile 3` only file 2 has changed. Compiling both of the files at this point would mean that file 1 is being compiled needlessly. Due to these unnecessarily large compilations, and IDEs continually building software, there is a higher workload when using a batch compiler. Furthermore, the time wasted in batch compilers will lower developers' productivity. Make offers a simple solution to this problem: it tracks file changes and only recompiles those which have changed or have dependencies to the modified code [6]. This strategy is called incremental compilation, and offers significant benefits to software development.

1.2 Incremental Compilation

Incremental compilers aim to reduce these workloads by compiling smaller sections of the program [7]. This reduction is typically achieved by recompiling only the changed portions of a program since the last compilation [4]. For example, when Rust developers asked Dropbox what they wanted from Rust in production, they answered "faster compiles" [8]. Rust has notoriously slow compile times, which without incremental compilation has been affecting developers' productivity. While Rust worked similarly to Make and avoided compiling the entire program, compilations were still too slow and Rust developers started working on an incremental compiler. The Rust incremental compiler works similarly, where "Incremental compilation avoids redoing work when you recompile ..., which will ultimately lead to a much faster edit-compile-debug cycle." [8] The Eclipse IDE already has an incremental compiler built-in; however, this is not much more sophisticated than Make [4]. The Eclipse incremental compiler only recompiles files with changes since the last compilation [4]. Surprisingly, Eclipse does not support more sophisticated approaches such as analysing whether a change will affect the program. For example, changing the line `i = i + 1` to `i += 1` is a small change to the overall program and typically would not require recompilation.

1.3 This Project

While incremental compilation by reduces the overall time of a compilation, there has not been research into its effect on compilation pipelines. For example, not much is known about the typical workloads of incremental compilers. Research into types of errors typically arising in programs, such as syntax, semantic and logic errors does exist [9, 10, 11, 12]. However, this literature does not investigate how these errors impact a compiler. For this report, we consider a workload as a sequence of edits made by a user. An important analysis of workloads is, if a workload is rejected by a compiler, why was it rejected? An example of this is identifying which stages of a compilation pipeline caused the failure. Identifying bottle-necks or common errors during these compilations would prove to be beneficial in compiler design, particularly for incremental and distributed compilers. For example,

in a distributed web compiler, such as VS Code, Theia, or Ellie, compilation could occur on either the client or the server-side of the web application. If the parsing stage finds errors frequently in a development session, it could be beneficial to move the stage to the client-side of the application. This shift to the client-side could catch parsing errors, without requiring the client to send information to the server, thus reducing the overall compilation workload and capturing the most frequent errors on the client-side. This project aims to analyse the compilation workloads in such a way as to discover how to improve incremental and distributed compiler design.

1.4 Contributions

There have been a number of contributions to this project. These include:

1. A web application was developed to enable users to complete programming challenges and to capture the results of their compilations. To enable this, the OpenJDK Java compiler has been instrumented to enable capturing compilation workloads. Additionally, a web interface has been developed in AngularJS, a server in Java using `org.apache.http` libraries, and PostgreSQL as a database. This report discusses the design and implementation of this application.

2. User studies were completed to capture compilation workloads throughout users' development sessions. A total of 19 users participated in the study, generating 1151 compilations. The results of the studies have been analysed and are presented through a number of graphs. Some key findings from the study are that a majority of errors occur in the *parsing* stage of compilation, but with a larger number of *type checking* errors when developers use complex type systems such as generics. Human Ethics approval from Victoria University of Wellington was required for this project and granted under the number 0000028477.

1.5 Covid-19 Effects on this Project

There have not been significant effects on the ability to run the user study for this project due to the user study being run after the country-wide lockdown in March. However, there has been significant effects to the implementation throughout this project, due to the pandemic. The uncertainty and changing timelines of the University year have significantly increased stress and decreased motivation for project and assessment work. Furthermore, as exams have been removed, the assessment timeline compressed, and additional assessments added to make up for the lack of exams, the amount of work required throughout the year has significantly increased. This has lead to a less than ideal amount of time to dedicate to this project, among others.

Chapter 2

Background

This chapter covers background surrounding incremental compilers and OpenJDK. This chapter also discusses several relevant techniques from the literature.

2.1 Incremental Compilation

Software compilation is the process by which a computer translates a program from code into instructions that are executable [13]. For example, the Java programming language compiles to Java byte code, which is then run on the Java Virtual Machine (JVM) [14]. Traditionally compilers work in a pipeline, which is essentially a sequence of consecutive jobs [15]. A pipeline ensures that the execution of these jobs occurs in a defined order. Specific jobs may depend on the outcome of previous stages, and the completion of prior tasks fulfils these dependencies before subsequent phases of the pipeline run [15]. For example, before a compiler can run checks and analysis, it must parse (process textual input) the program [7].

There have been implementations of incremental compilers since the 1960s [7]. Early incremental compilers were more effective than current incremental compilers due to the simplicity of programs at the time. For example, incremental BASIC was one of the first incremental compilers in 1968 [7]. Programming languages such as BASIC were low level languages and were similar to assembly code. Due to the low level nature of these languages, incremental compilers were trivial for them as they only required recompiling changed lines. As programming languages became complicated with more language features, developing incremental compilers for them became significantly more difficult.

As an example to illustrate, in 2016, the Rust programming language had an incremental compilation option added to its compiler [8]. The Rust incremental compiler tracks dependencies through multiple stages of compilation. This tracking allows the compiler to update cached data from previous compilations for only the segments which require updating. Analysis of the performance of the Rust incremental compiler shows slightly slower initial compile times due to the caching required. However, further recompilation shows dramatically reduced compilation times for most cases. This 2016 release of the Rust incremental compiler is only an alpha version, and there is plans for significant work before a fully incorporated incremental compiler will be completed. In the 2018 release of Rust, incremental compilation is enabled by default. However, there is still a large amount of work planned for incremental compilation in Rust [16].

2.2 OpenJDK Compiler Pipeline Overview

Compilers are traditionally structured in a pipeline with two sections relating to the main goals of the compiler. The frontend and backend of a compiler confirm the code is correct

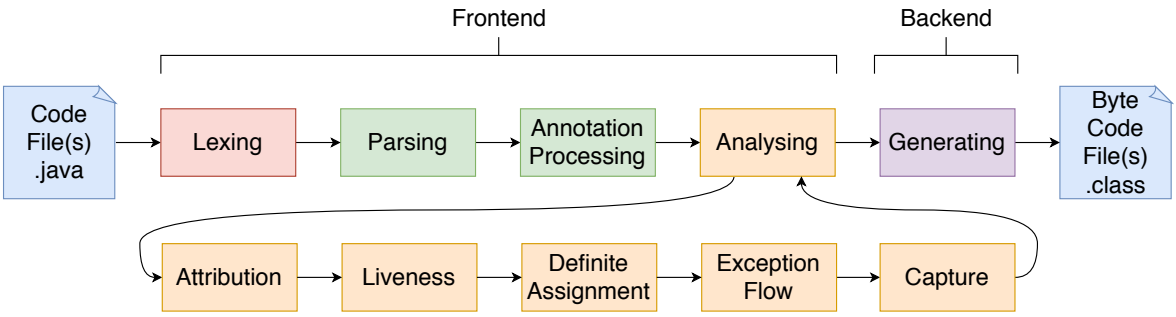


Figure 2.1: OpenJDK Compiler Pipeline

and generate executable files, respectively. The frontend typically creates an abstract syntax tree (AST) which represents the program in a structured way, with only relevant information. The AST is then checked in a number of passes during the *analysing* phase of the compiler. Finally, the backend of the compiler converts the AST into executable binary or byte code files.

As seen in Figure 2.1, there are five main phases in the OpenJDK compiler pipeline. These phases are *lexing*, *parsing*, *annotation processing*, *analysing*, and *code generating* [17]. The *lexer* reads a source code file and converts it to a stream of tokens, which include variable names, operators, keywords, etc. The *parser* converts the token stream into an AST. *Annotation processing* is an iterative extension of the parser in which a developer can specify particular behaviours from the compiler [17, 18]. *Analyse* performs a number of checks on the AST to ensure the code is correct and to generate informative error messages when the code is incorrect. Finally, the *generate* phase converts the checked AST to Java byte code, which can be run on the Java Virtual Machine (JVM) [14].

2.3 OpenJDK Compiler Analyse Phase

The *analyse* phase in Figure 2.1 shows a breakdown of six analyses performed on the AST. The *attribution* analysis checks variable names and type information [17]. This check includes ensuring variable types match throughout the variables' scopes. For example, take a variable of type String: this check ensures that such a variable is never assigned a different type. Listing 2.1 shows a (correct) program where a variable of type String is assigned as a String twice. Contrastingly Listing 2.2 shows another (incorrect) program where the variable is assigned as a String and then as an Integer. The name resolution checking that occurs in this phase ensures that a variable which is used exists in the program. Listing 2.3 shows variable `x` being assigned, however it has not been declared in the program and will fail name resolution.

```
String testString = "this is a string";
testString = "this is a different string";
```

Listing 2.1: Correct Type Assignment

```
String testString = "this is a string";
testString = 1;
```

Listing 2.2: Incorrect Type Assignment

```
int i = 0;
```

```
x = i + 2;
```

Listing 2.3: Failed Name Resolution

The *liveness* analysis performs a dead code analysis on the AST. This analysis searches for unreachable code and raises an error if it finds any statements which are unreachable. An example of unreachable code is any code listed after a block's exit point. Listing 2.4 shows this: the statement `int y = 2;` comes after the `return;` exit point.

```
f() { int i = 1; int x = i + 2; return; int y = 2; }
```

Listing 2.4: Unreachable Code

Definite assignment analysis ensures that variables are defined before they are used in a program [19]. Listing 2.5 shows a program in which a variable is declared before its use, but is not assigned. This program fails definite assignment checking as the expression `int x = i + 2;` attempts to evaluate `i` before it is assigned.

```
f() { int i; int x = i + 2; i = 1; }
```

Listing 2.5: Variable Use Before Assignment

The *exception flow* analysis follows exception propagation paths to ensure that checked exceptions which are thrown in the program are caught. Listing 2.6 shows an `IOException` being thrown. In Java, these are checked exceptions and must be caught to ensure they do not propagate to the root of the program. In Listing 2.6, the exception is not caught in the `main` method and will propagate to the program root. To fix this error, the exception must be explicitly declared by the `main` method, such as in Listing 2.7 or caught in a `try-catch` block, such as in Listing 2.8.

```
public static void main(String args[]){
        if (args.length < 3) { throw new IOException(); } ...
}
```

Listing 2.6: Uncaught Checked Exception

```
public static void main(String args[]) throws IOException {
        if (args.length < 3) { throw new IOException(); } ...
}
```

Listing 2.7: Declared Exception

```
public static void main(String args[]){
try{ if (args.length < 3) { throw new IOException(); } }
catch (IOException e) { ... } }
```

Listing 2.8: Caught Exception

Capture analysis checks that local variables in lambda bodies or local inner classes are final or effectively final. Listing 2.9 shows a variable `i` being assigned from within a lambda body. This assignment will fail the *capture* analysis, as `i` should be final and its value not change within the lambda body.

```
int i = 2; stream.forEach(int item -> i = item;);
```

Listing 2.9: Non-Final Lambda Body Reference

2.4 Related Work

We now examine a selection of papers from the literature on incremental compilers.

Responsive Compilers

Nicholas Matsakis is a significant contributor to the Rust programming language and gave a presentation about incremental compilation in Rust at PLISS 2019 [20]. Matsakis highlights how interactions with compilers have changed due to the interactivity of IDEs. When a developer makes a small change in an IDE, they expect fast feedback about the compilation of that change. The Rust compiler (rustc) uses the language server protocol (LSP) as an intermediate interface between a language and an IDE. The LSP abstracts calls between the two, and reduces the need for multiple interfaces for different language-IDE combinations. In the current IDE-compiler architecture, the compiler becomes an actor.

Rustc uses a framework called *salsa* to implement the compiler. Salsa is based around queries, which are pure functions used to discover what other information and dependencies the compiler need. Results from these queries are cached so that the cached result can be returned instead of re-evaluating the query. This is similar to the abstract language used in PECAN discussed in *Generating an Incremental Compiler*.

An internal database stores information about derived salsa queries. The database contains a revision counter and a map for each query. The query map contains a cached result for the latest revision, dependencies and the revision in which the last change was made. When a query is invoked, if there is no cached result the query must be computed. Otherwise, if the dependency is out of date, the compiler updates the query and latest revision.

Rustc can use these derived queries, and query maps for incremental compilation. Revisions for AST nodes are only updated when the new result is different from the old result. If the input file is updated, its revision number is increased. When the file is parsed to an AST, if the AST is unchanged for that file, the revision number is not increased. In this case, subsequent phases can simply use the cached results from the previous compilation.

Incremental Compilation for Continuous Integration

Guillaume Maudoux and Kim Mens discuss the possibility of applying incremental compilation to continuous integration (CI) pipelines [21]. CI is the process of automatically building and testing software on a repository check-in. The aim of the pipeline is to ensure that the software is safe for deployment. The issue in applying incremental compilation to CI is that the pipeline typically runs in a fresh environment from clean sources. This makes incremental builds impossible, as there is no saved information from previous compilations.

Madoux and Mens claim that CI build environments could be improved using incremental compilation. An analysis of compilation times for the Mozilla Firefox web browser show that a large number of commits result in no change to the software. For these cases, a full rebuild would be pointless and result in time wastage. For commits where the source code changes, the median build time was approximately 20 times quicker than a full rebuild. Based on their experiment, about 90% of CPU time on CI workers is wasted, and incremental builds could significantly improve performance and resource use.

The main barriers to adopting incremental compilation for CI environments include: concerns that existing incremental compilers do not build software correctly, impurities may be introduced to build environments due to caching of incorrect builds, and workers may not have access to build results from other workers.

The proposed solution is to use the 'correct' algorithm from *tup* and the *nix* package manager to implement an incremental build system with a shared cache between CI workers. *Tup* is a build system which uses a list of file changes and a directed acyclic graph to update dependent files [22]. *Nix* is a package manager for Unix systems which isolates packages to ensure software builds are reliable and reproducable [23]. Madoux and Mens caveat

the proposed solution by acknowledging that accessing the cache could be slower than a local rebuild. Finally, they conclude that a "huge speedup in compilation time could make [the incremental build system] attractive."

Achieving Incremental Compilation through Fine-grained Builds
Tim Cooper and Michael Wise describe *Make()*, a procedure-level build tool which they developed [24]. Make() achieves incremental compilation by processing source code by procedures instead of the entire source code. Build tools process dependencies between subsections of source code. A traditional build tool is *make*, which operates on the file-level, managing dependencies between files. Make() expresses dependencies between objects, such as procedures, types, global variables and macros. Make() is built for the C++ language, with some language features omitted. Make() is intended to scale for large projects and the methods described to be applied to other statically typed languages.

An optimisation for dependencies is discussed, but not implemented. In this optimisation, an object's interface is separated from its body. The interface is the surface of an object, which can directly affect another object. This typically means the prototype for a function. In this interface optimisation, dependencies only require updating if the interface changes. This could mean the body of a function is updated, and any object calling the function doesn't require updating because the interface remains the same.

Make() handles recursive dependencies by marking objects as visited the first time they are recompiled. This prevents expensive computation and infinitely recursive compilation. However, dependencies may still require updating after the first pass, and so a number of passes may occur to ensure the compilation is up to date.

Make() is shown to be significantly more efficient than *gcc* and *Visual C++*. For example, to add a comment to the central header file for a project, 139,736 lines of code required updating in gcc, 34,926 in Visual C++, but only 4 in Make(). This improvement occurs, as file-based build tools will recognise the central header file has changed and will rebuild all dependent modules. In Make(), all procedures in the central header file are recognised to be unchanged, and only a comment object needs to be added. There are also good, albeit not as significant improvements when rebuilding all modules (150,231 for gcc, 21,320 for Visual C++, and 20,159 for Make()).

Build System with Lazy Retrieval for Java Projects
Molly is a build system which has lazy project dependency retrieval [25]. Traditional build systems, such as Maven, typically require retrieval of all project dependencies prior to executing other build steps such as testing. In CI pipelines, dependency retrieval can be significantly time consuming, with some projects taking up to an hour to retrieve all dependencies.

Molly tightly couples the compiler, runtime, and build system and retrieves dependencies lazily during execution of build targets. When a library is added to the Molly Central Repository, the library is *split* into files, and *trimmed* to create a public API. This provides a stripped-back version of the dependencies which can be used in compilation without the unnecessary bloat of the full libraries. While the full dependencies are required for testing, if only a build is occurring the content of these dependencies is not required. When the software is run, Molly replaces the trimmed dependencies with the required files from the library. Only using the required files reduces the size of the dependencies, as the whole library is not retrieved.

Molly can also cache built class files for incremental CI builds. Merges are performed to load cached class files and for replacing trimmed libraries with the required files. The merge process is entirely transparent to the user, but is expensive. Because of this, an asynchronous merge is implemented, which can replace trimmed libraries as a background process.

The overall system is shown to reduce dependency retrieval time by 44.28%, reduce disk

space use by 89.80%, and reduce retrieval time in standard cases by 93.81%.

Incremental Whole Program Optimisation and Compilation
Whole program analysis and optimisation is typically performed in modern compilers for high level languages [26]. While this approach provides the compiler visibility to the entire program, it makes recompilation a computationally expensive task, where the whole system must be recompiled. An approach to whole program analysis and optimisation has been incorporated into the commercial Visual C/C++ compiler.

In the existing compiler, whole program analysis (WPA) runs on a single thread and takes approximately 30% of compile time. The incremental version runs WPA incrementally across functions in the program. Functions which are changed and dependencies are recompiled and WPA applied to these. If the current code is changed, then the function requires recompilation.

To ensure incremental builds have the same output as a non-incremental build, the compiler must balance incremental efficiency and optimisation. Comparisons are drawn between functions whose dependencies have been updated, but whose code has not changed, with regard to the dataflow throughout the program. If the function's dataflow is found to have moved in a 'downwards' direction, as determined by an implemented algorithm, the function must be recompiled. However, if the dataflow moves in an 'upwards' direction, the generated code could be improved if recompiled, but is not required to be correct. This analysis results in some functions being recompiled even if their source code has not changed, in order to ensure that correct optimisation is achieved in the system.

The system was tested by building a number of libraries and executable files. Through these tests, the system was shown to compile more than seven times faster than non-incremental builds of the same artefacts.

Generating an Incremental Compiler
Steven Reiss describes an approach to generating an incremental compiler [27]. PECAN development environments make the use of graphical user interfaces (GUIs) and are some of the first integrated development environments (IDEs). The PECAN incremental compiler uses an abstract language to allow incremental compilation for a number of languages.

The incremental compiler designed for PECAN consists of five modules. The primary module stores a linear sequence of expressions generated from the AST. This module is responsible for updating the linear sequence by catching change messages for the sequence. Updating the sequence is completed by rolling back the actions from the existing sequence, then playing the new actions on the sequence. The rest of the modules provide entries for execution and rollback of actions for their subset of the linear representation. These modules propagate any changes made to them through to the rest of the compiler.

The primary module of the incremental compiler updates the actions when the AST is updated. When updating the actions, the compiler creates a list of actions for the change, compares the list with the previous list of actions to find where it should be inserted, rolls back the old list to the start point of the new changes, applies the new changes and finally, executes the following actions to complete the compilation.

A number of actions occur when updating a compilation. First, the symbol table is updated, then data types, expressions, and finally control flow. All changes in definitions are resolved and the properties of all references to the definition are updated. When a symbol changes, the rest of the compiler is sent a message to inform that the reference has been changed and the changes that were made.

The approach was implemented in C and was used as a part of a programming environment. Nothing other than the semantic specifications are required and the approach has been shown to be efficient.

Chapter 3

Design

This project aimed to investigate how interactions between a user and a compiler happen at a fundamental level. In particular, we examine edits to code produced by real users and analyse how the compiler processes these edits and which compilation stages were involved. The information generated from this analysis can be used to inform incremental compiler design and development. Similarly, it will offer insight into the process of programming. We developed a simple web application for writing and compiling Java code and used the application to perform a user evaluation. This chapter discusses the design of this tool, including the architecture and some alternative approaches considered.

3.1 User Study

The aim of the user study was to generate compilation workloads which one could imagine occurring in a standard development session. A standard development session would be a period of time in which a developer works on a project as they usually would in a workplace. These challenges were completed in the web application created as a part of this project. A variety of challenges have been created which attempt to replicate problems which may arise in a development session. These challenges were designed as small algorithmic or Java language puzzles which require the user to complete an algorithm or use a particular language feature. Due to the small nature of the challenges, we do not expect the results to be entirely representative of a standard development session. However, we believe the challenges offer a reasonable attempt at replicating these sessions within the constraints of this project. The challenges created include:

1. Counting the number of occurrences of an item in an array (**Count**).
2. Implementing a class hierarchy (**Covariance**).
3. Writing programs with exceptions and handling exceptions (**Exception Handling** and **Exception Handling 2**).
4. Flattening a binary tree into an array (**Flatten**).
5. Exercises which require defining and using generic and wildcard types (**Generics** and **Generics 2**).
6. Finding the maximum of three numbers (**Max 3**).
7. Finding the permutations of a string (**Permutations**).
8. Reversing an array (**Reverse**).
9. Searching for a subarray within a larger array (**Search**).
10. Sorting an array (**Sort**).

Each challenge is comprised of a number of components. The challenges each have a name, a set of instructions, some template code, and a test suite. The instructions inform the user of what they are trying to achieve in the challenge, and the template code provides a starting

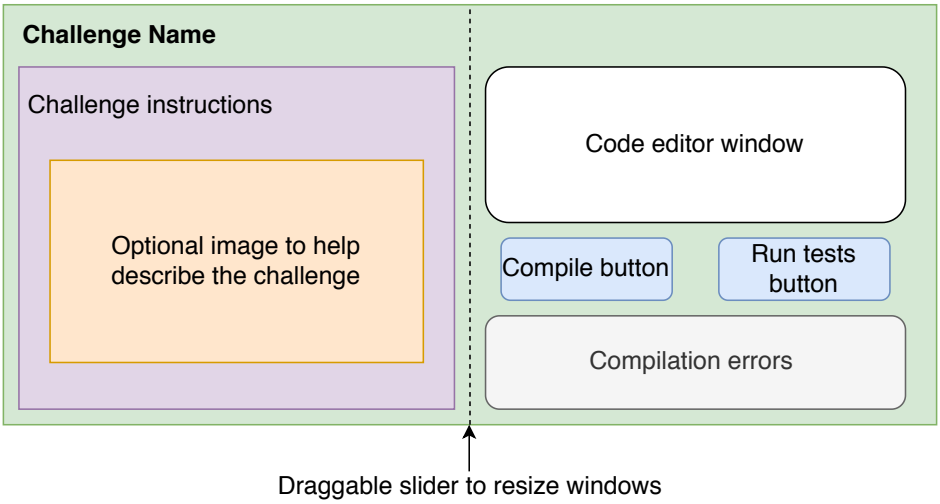


Figure 3.1: Mock-up of the Web Interface Main Page

point. Section 3.1.1 discusses how this information is presented to the user. When completing a challenge, a user writes code and presses the compile button. The user's code is then compiled and the user receives feedback as to whether the compilation succeeded or not. If the compilation succeeds, the user can run the test suite associated with the challenge to check if their code completes the challenge.

3.1.1 Web Interface

Figure 3.1 shows the design of the main page of the web interface. There are two main panes on the page, which are separated by a draggable slider. The left pane has the instructions for the current challenge and an optional image which could help clarify the instructions. The right pane has a code editor, buttons for compiling the code and running tests, and a window for displaying compilation errors. The draggable slider allows the user to move the challenge instructions out of the way so they can focus on their code if they wish.

3.2 Architecture

The web application developed enables compilation and collection of the results of Java programs written by users during the user study. The overall architecture of the system is shown in Figure 3.2. The web server is the main part of the application, with the web interface, database, and OpenJDK components loosely coupled to the server. The OpenJDK component consists of two main sections, the compiler is used to compile the user's code, and the runtime is used to run tests on compiled code.

3.2.1 Compiling a Program

Figure 3.3 shows the workflow for a compilation request in the developed web application. There are a number of steps to compile a Java program:

1. A user writes a program in the web interface and requests compilation of the program by pressing the `Compile` button. The web interface sends the user's code over a HTTP POST request to the `/compile` API endpoint on the web server.
2. The web server reads the code sent from the web interface. The server adds any required imports and wraps the user's code in a public class to enable compilation. The

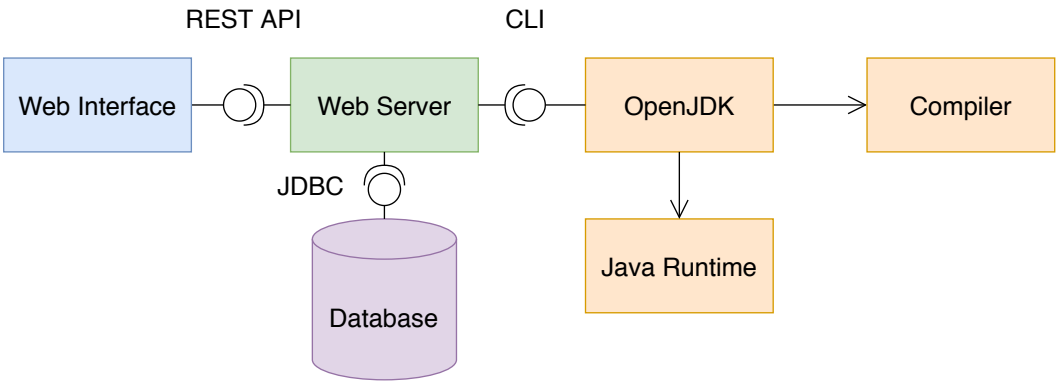


Figure 3.2: System Architecture

server then writes the code to a file. The file's name matches the public class which the code has been wrapped in.

3. The Java compiler is used to compile the program. The compiler used is based upon the OpenJDK Java compiler. We have altered the compiler to generate instrumentation output which indicates the compilation stages which are passed or failed.
4. The Java compiler outputs the Java byte-code generated from the compilation to a class file.
5. The web server parses `stdout` and `stderr` from the compiler process to find any compilation errors and which stages of the compilation have passed or failed.
6. The web server stores the compilation information in a PostgreSQL database. The information stored includes the user's code and the stages of the compilation which have passed or failed.
7. The web server responds to the `/compile` request with *Compilation Successful* if the compilation succeeded. If the compilation failed, the server responds with the errors which occurred in the compilation.

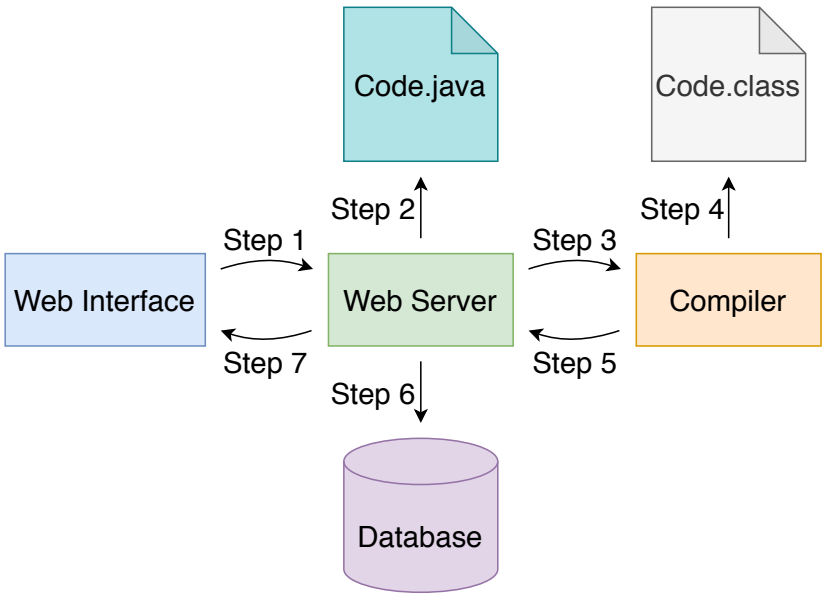


Figure 3.3: Workflow of a Successful Compilation

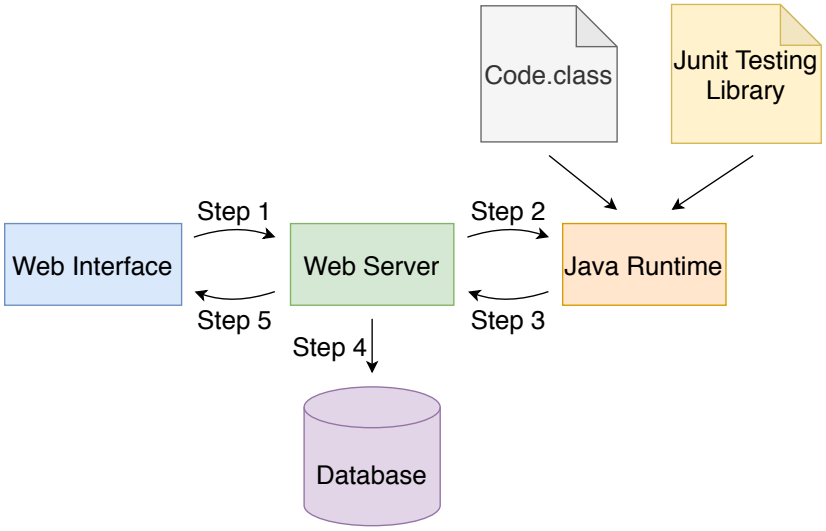


Figure 3.4: Workflow of a Testing Request

3.2.2 Testing a Compiled Program

Figure 3.4 shows the workflow for a user running tests on a successful compilation. The web server dispatches the test request to the Java Runtime with use of a Junit testing library to test the compiled code. The test results are stored in the database and are sent back to the web interface. There are a number of similar steps involved in testing a compiled program:

1. After a user has successfully compiled a program, they can press the `Run Tests` button. The web interface sends an HTTP POST request to the `/test` endpoint on the web server. This request includes a session key as a unique identifier for the user, and the challenge which the user is working on.
2. The web server dispatches a command to the Java runtime, including a Junit testing library and the compiled class file in the classpath. This command runs the test suite associated with the specific challenge the user is working on in a sandboxed environment for security.
3. The web server parses `stdout` and `stderr` from the runtime process to find the number of successful and unsuccessful tests.
4. The web server stores the test results in the PostgreSQL database. The information stored includes an ID relating to the compilation to which the test request relates, the session key, and the number of passed tests.
5. Finally, the web server responds to the `/test` request with the number of tests passed and failed. The web interface displays this information to the user.

3.2.3 Application Components

Web Server

The web server provides a central interface for all the logic and actions required to implement the system. The other sections of the system are connected to the web server and are loosely coupled to ensure that any of the other components can be replaced easily. A REST API is used to receive requests from the web interface. The command line interface (CLI) is used for compiling code and running tests. Finally, the Java Database Connection (JDBC)

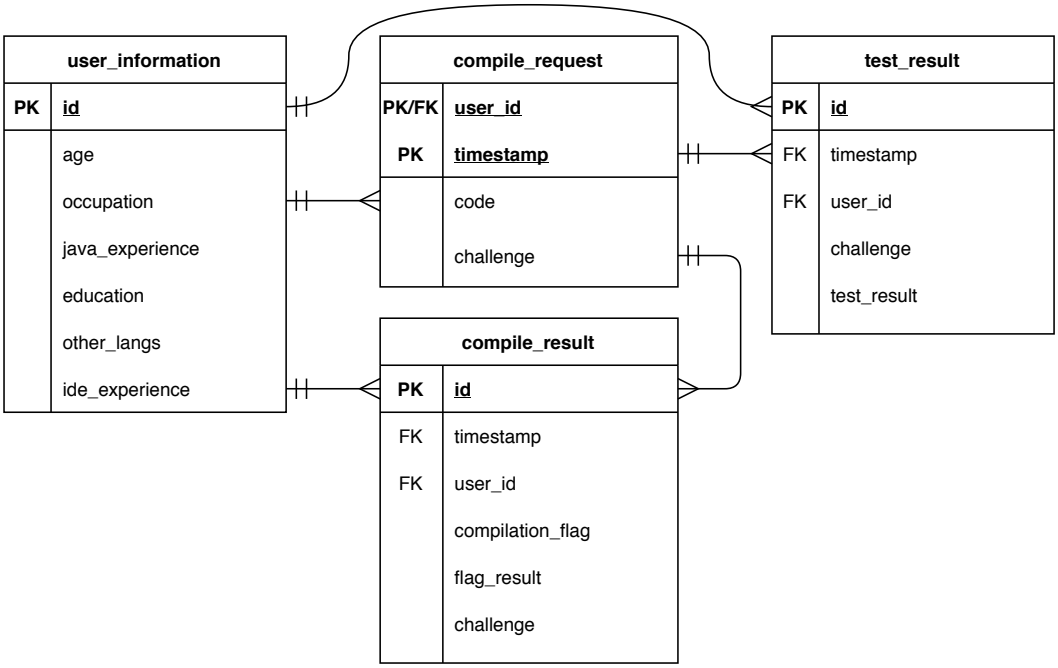


Figure 3.5: Database Entity Relation Diagram

library provides the abstracted connection for the database. The database and compiler connection is discussed further in the following sections.

Database

Connecting to a database should not be a hard-coded process. Abstracting connections to the database enables the specific database instance used to be replaced. This loose coupling allows programs to use multiple different database servers within a single program, or to easily replace one database server with another.

The database provides storage for information the user provides, their compile requests, and the outcomes of their compile and test requests. There are four tables in the database which represent different aspects of the data generated in the system. Figure 3.5 shows the tables and relationships between them. The data in all the tables is directly linked to a user id. Due to database foreign key constraints, this means that data cannot be added to tables other than `user_information` without a specific `user_id` existing in that table.

The `user_information` table stores information the user enters in a form on the web interface. The non-primary fields in this table are information which the user enters as a part of the user study. Collecting contextual information, such as the users' occupation and experience helps to draw more conclusions about the results of the user study.

The `compile_request` table holds information about a user's compilation request. Importantly, it holds a timestamp of the compilation, the code for the compilation, and the name of the challenge the user is working on. The `user_id` field is used in the primary key, along with the `timestamp` to ensure the uniqueness of each record in the table. The `code` field holds the code which a user has requested to compile, and `challenge` is the challenge name for the challenge which the user is completing.

The `compile_result` table holds the instrumentation generated by the compiler. The name of the challenge which the user is completing is stored. Additionally, the compilation stages and results generated from the instrumentation in the compiler are stored. Separating

the compile request and result tables enables conclusions about compilations to be drawn for the whole dataset without requiring information about each specific compilation.

The `test_result` table contains the outcome of a testing request. The information stored includes the number of tests passed and the total number of tests run on the user's code.

Compiler

Calls to the OpenJDK from the web server occur through the command line interface (CLI). The web server specifies the CLI command to run, such as `javac Code.java`, and runs this command as a user would in the CLI. The process is watched by the server and the output of the process are collected. The server binds to the `stdout` and `stderr` to collect compilation instrumentation, compilation errors, and test results. This abstract connection through the CLI enables the specific OpenJDK version (including the compiler and runtime) to be replaced with minimal effect on the server.

3.3 Alternative Designs

A proposed alternative to the implemented system was considered but disregarded as unsuitable for the project. This approach would have been to use the altered OpenJDK compiler as the compiler in a preexisting IDE such as Eclipse or IntelliJ on the ECS machines at the University. The main advantage of this approach is that it would have enabled users to use all the tools these IDEs provide, such as debuggers, auto-complete features, and the ability to write their own tests for the challenges they are asked to complete. There are, however, a number of disadvantages to this approach. While incorporating the altered compiler into an IDE would work, this approach would require ensuring that the altered compiler were accessible on the ECS machines. It would be difficult to find users to take part in the study as they would be required to be present at the University to take part in the study. This restriction would likely restrict the users to students, with few industry developers being able to take part. Additionally, this approach would require configuring the IDEs prior to the user studies to ensure the correct compiler is used. Another issue is that the compiler could be accidentally used for the compilation of projects not included in the user study if the IDE were not reconfigured back to its original settings after the study. Furthermore, given the restrictions on gathering due to Covid-19, it was decided that a web application would be a better approach with no gathering required. It is also easier to include a more diverse population for the user study.

Another design which was disregarded was for a more comprehensive user study. Instead of completing small coding challenges, which are not extremely representative of standard development, assessing workloads throughout development on a larger system was discussed. While this would result in a more in depth and representative dataset, such a study would require a more longitudinal approach. Due to the timeline restrictions of this project being an honours project, this approach was disregarded as unachievable in the timeframe.

Chapter 4

Implementation

This chapter describes the implementation of the web application for capturing compilation workloads. The application has been implemented based on the design discussed in Chapter 3. There are a number of components in the web application, including the Java compiler, coding challenges, web server, web interface, and the database. Furthermore, there are a number of other ideas discussed such as concurrency, security, and handling long running processes.

4.1 Java Compiler

The Java compiler (Javac) used is an altered version of the OpenJDK Java compiler, which implements Java 15. OpenJDK is an open-source Java Development Kit, which includes a Java compiler for compiling Java programs, and a Java Runtime to run compiled Java programs. Javac is a large program, with the source code consisting of 15 packages, 187 files, and 128960 lines of code. Instrumentation has been added to record which pipeline stages are reached during a compilation. This instrumentation provides key information about which stages of compilations pass and fail. The instrumentation added to the compiler mostly matches the phases described in Sections 2.2 and 2.3. These are *parsing*, *attribution*, *liveness*, *definite assignment*, *exception flow*, and *capture*.

The implementation of instrumentation in Javac has been achieved by printing messages to the console. As discussed in Section 3.2.3, the web server attaches to compiler processes using `stdout` and `stderr`. In Java code, this translates to method calls of `System.out.println` and `System.err.println`. The instrumentation added to the compiler is printed only to `stdout`, leaving `stderr` free for compilation error messages generated from the compiler. Listing 4.1 shows a simple example of how the instrumentation has been added to the source code of OpenJDK. This example is for *capture*, where the number of errors already raised in the compiler are saved to the temporary variable `prevErrors`. The line `scan(tree);` performs the capture analysis on the current AST which will increase the error count if errors are raised. The `if-else` statement compares the number of errors before the capture analysis to the number after. If there are more errors after, the string stating the analysis failed is printed to `stdout`, otherwise the string stating the analysis is complete is printed.

```
public void analyzeTree(Env<AttrContext> env, JCTree tree,
    TreeMaker make) {
        int prevErrors = log.nerrors;
        scan(tree);
        if (prevErrors < log.nerrors) {
            System.out.println("Flag - Capture: Failed");
        } else { System.out.println("Flag - Capture:
            Complete");
        }
}
```

Listing 4.1: Implementation of Compiler Flags

4.2 Coding Challenges

The coding challenges developed cover a number of language features and techniques as discussed in Section 3.1. The challenges have been developed using Java 15 so they can be used with the compiler discussed in Section 4.1. Each challenge includes a Markdown file of the instructions, a Java file with template code, and two test files. Some challenges also include images which clarify the instructions, and a file with the list of imports allowed for the challenge. For example, Figure 4.1 and Listing 4.2 show the instructions and template code for the *Count* challenge. Of the two test files, one file contains the actual tests which should all pass when the user completes the challenge correctly. The other test file contains very basic tests which ensure that the user is compiling their code against a skeleton for the class or method signatures. Compiling against the skeleton cases ensures that the user's code will run with the actual test cases. Furthermore, any error messages from the compiler which arise when the user compiles against the skeleton code will not have information about the logic of the tests. This ensures that the user receives error messages if their code will not work with the test cases, while also ensuring the user is not inadvertently given the solutions for the challenges. When the user runs the tests for a challenge, their code is run using the actual test case. Compiling against the skeleton code initially ensures the code does not require compiling again as it will already have the required method or class signatures. Therefore, the tests can be run and the user will receive information about the number of test cases which pass.

```
public static int count(int[] arr, int item) {
    return 0;
}
```

Listing 4.2: Count Template Code

Listings 4.3 and 4.4 show the difference between test cases in the main and skeleton test files. As can be seen in Listing 4.3, there is the body of an entire test case. An array is created and populated with integers between 0 and 15. The expected count value is calculated in a helper method call: `count(arr, countInt);`. The actual count calculated from the user code occurs in the static method call: `Count.count(arr, countInt);`. Finally, an assertion is called to check that the expected and actual count values are the same. If they are the same the test passes, otherwise the test fails. The skeleton test case in Listing 4.4 differs from the main test case, as it has no assertion or logic in it. The case only has the call to the user's

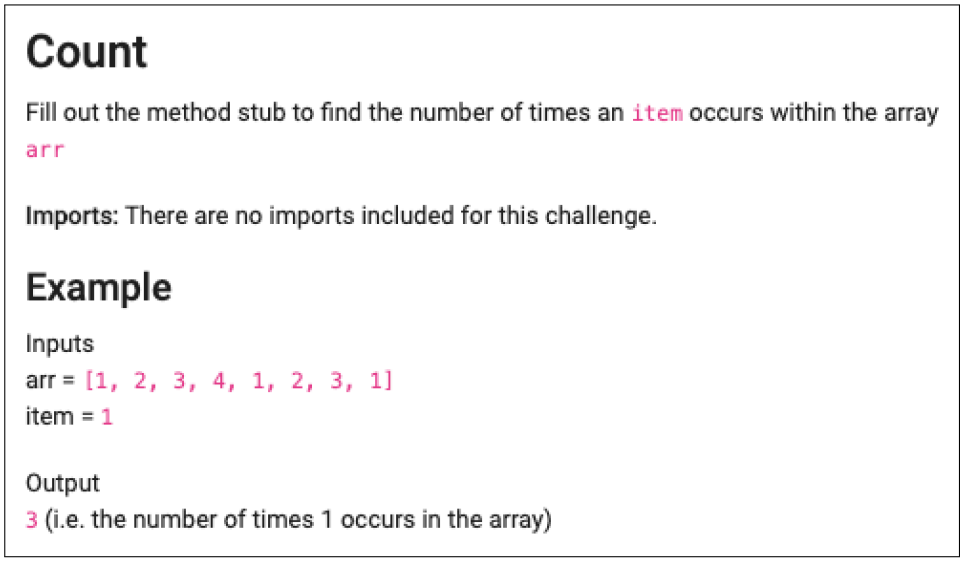


Figure 4.1: Count Instructions

code. This skeleton test case requires the user's `count` method signature to match the template code provided. In this case, from the skeleton test case, we can tell that user's method must be static, called `count`, take parameters of types `int[]` and `int`, and return an `int`.

```
int[] arr = new int[100];
for (int j=0; j<100; j++) {
        arr[j] = r.nextInt(15);
}
int countInt = r.nextInt(15);
int expectedCount = count(arr, countInt);
int actualCount = Count.count(arr, countInt);
assertEquals(expectedCount, actualCount);
```

Listing 4.3: An Example Count Test Case

```
int actualCount = Count.count(new int[5], 1);
```

Listing 4.4: An Example Count Skeleton Test Case

4.3 Web Server

The web server provides the core functionality of the application and connects the other components of the system together. As discussed in Section 3.2.3, the web server provides loosely coupled interfaces for the web interface, OpenJDK, and database. The web server itself is written in Java and uses `org.apache.http` libraries to implement the server functionality such as exposing endpoints and persistent execution of the server.

4.3.1 Server Actions

There are a number of API endpoints provided by the web server. These endpoints are used to specify actions in a simple, understandable format. These include *challenges*, *compile*, *test*,

storeUser. Additionally, the server also provides handling for static `html`, `js`, `css` files. Finally, the server also provides an endpoint for images included in challenge instructions. Each of these endpoints is implemented as an `HttpRequestHandler` object for ease of use with the apache libraries. All requests and responses with bodies are JavaScript Object Notation (JSON) encoded. JSON provides a simple and understandable format for encoding web request and response bodies. An example of JSON is provided in Listing 4.5.

```
{ 'id': 'a user id', 'data': ['an', 'array'] }
```

Listing 4.5: A JSON Object

Challenges Endpoint

The *challenges* endpoint provides a handler to return the coding challenges described in Section 4.2. This handler builds a JSON array of the challenges. Each item in this array is a JSON object including the challenge name, starter code, and instructions for the challenge. Additionally, due to the challenge instructions being written in Markdown, the instructions will not render correctly in a web format. Before the instructions are stored in the JSON object, they are compiled to HTML. `org.commonmark` libraries are used to compile the Markdown instructions. Initially, the Markdown is parsed using `org.commonmark.parser.Parser`, and then `org.commonmark.renderer.html.HtmlRenderer` is used to render the parsed file to HTML. This enables a web interface to simply display the instructions in a browser and they will display correctly.

Compile Endpoint

The *compile* endpoint receives compile requests which include a JSON object request body. Listing 4.6 shows an example of a compile request body. The `sessionKey` is a random UUID generated in the web interface which uniquely identifies a user of the application. The `challengeName` is the name of the challenge which this compile request relates to. Finally, `code` is the user's code which they are compiling.

```
{
        'sessionKey': 'user id hash',
        'challengeName': 'Count',
        'code': 'public static int count(...) { ... }'
}
```

Listing 4.6: Compile Request Body

When the server receives a compile request, a directory path is constructed based on the session key and challenge name. A user's compilations are stored under their session key, thus creating the file path: `sessionKey/challengeName`. If the required directories do not exist, the server creates them and copies the imports and testing files into each directory. The server constructs the contents of the code file for compilation by including the contents of the imports file (if one exists), a package declaration, and wrapping the user's code in a public class. The name of the public class is the challenge name, which in turn is the name of the file to which the code will be written. Once the server has written the contents to the code file, a process is created to compile the code file and the skeleton test file described in Section 4.2.

The server reads the output from the process on `stdout` and `stderr`. Instrumentation which has been written to `stdout` by the compiler is parsed and added to the database.

This requires creating a timestamp to serve as part of the database ID for the compilation. The error lines are read from `stderr`. Error lines are edited to ensure the line numbers in the messages are not offset due to the extra lines of code added to wrap the user's code for writing to the file. Additionally, the line numbers which have errors are added to an array. If there are no error lines, 'Compilation Successful' is returned.

```
{
        'compileResult': 'Compilation Successful',
        'errorLines': [],
        'timestamp': 'timestamp'
}
```

Listing 4.7: Compile Response Body

Finally, the server constructs a JSON object to use in the response body to the HTTP request. Listing 4.7 shows an example response JSON object. `compileResult` holds the result of the compilation. This is either 'Compilation Successful' or the error messages written to `stderr` by the compiler. `errorLines` is an array of the lines which contain compilation errors. Finally, `timestamp` is the timestamp of the compilation.

Test Endpoint

The test endpoint is similar to the compile endpoint discussed in Section 4.3.1. Listing 4.8 shows an example of the request body for a test request. The `sessionKey` and `challengeName` are the same as those for the test request. The `compileTimestamp` is the timestamp generated during a compile request and returned from the compile endpoint. This timestamp is used to ensure the code which is tested matches the compilation which has occurred.

```
{
        'sessionKey': 'user id hash',
        'challengeName': 'Count',
        'compileTimestamp': 'timestamp'
}
```

Listing 4.8: Test Request Body

Similar to the compile request, when a test request is received the server builds the directory path based on the `sessionKey` and `challengeName`. The server compiles the test file to ensure that it can be correctly run against the user's compiled code. If the compilation succeeds, the server starts a process to run the tests. The server then adds the test results to the database and responds to the request. Listing 4.9 shows an example of the JSON response body.

```
{
        'compileErrors'?: 'Errors from compiling the test
           class',
        'testResults'?: '4/6'
}
```

Listing 4.9: Test Response Body

Both `compileErrors` and `testResults` are optional keys in the response. If the test class compilation fails the `compileErrors` key is set. This case should not happen due to compiling the user's code against the skeleton test cases as described in Section 4.3.1. However, this flag exists to ensure any cases which bypass the skeleton test compilation are caught before running the tests. Once the tests have been run the `testResults` flag is set with the number of tests passed over the total number of tests. This is used to inform the user if their solution to the challenge is correct.

Store User Endpoint

The `storeUser` endpoint allows both POST and DELETE HTTP requests and is used to handle a user's session on the server. When the user completes the information form described in Section 4.4 the server receives a POST request with the information the user has entered. This endpoint adds the user's information into the database. When the server receives a DELETE request, the server checks if the user directory exists on the system and deletes it. This ensures that there are no unused directories and code stored on the system as they are not needed once the user has completed their challenges. DELETE requests are sent from the web interface when a user finished the challenges, or closes the tab with the application open. There are some scenarios where a DELETE request would not get sent, such as if a user's computer crashed, or they turn off their computer without exiting the application. In these cases, the directories would not be deleted. However, this is a relatively minor issue and not a cause for concern here. In particular, during the user study, the deployment server was manually checked periodically to remove any stale directories.

4.3.2 Non-Functional Considerations

There were a number of considerations involved when implementing the server. These include concurrency, security, and process and connection orchestration.

Concurrency

Concurrency considerations were required to ensure the system would run robustly when multiple users are completing challenges at the same time. The `org.apache.http` libraries automatically handle concurrent requests and spawn a new thread for each request, ensuring that there is no bottleneck or timeout on these HTTP requests. The directory paths for users discussed above are a method of ensuring concurrent users do not have interfering processes when writing to files, compiling code, or running tests. A singular user could attempt to break the system by sending concurrent compilation requests to the server under the same user id, however this would likely involve some malicious scripting and was not regarded as a major point of concern. Storing a user's code files under their session key ensures that files created will not be accessible by users with different session keys. Additionally, writing information to the database occurs concurrently. When compiling code or running tests the user doesn't need to wait for the compilation flags or test results to be written to the database, as these do not affect their experience. Implementing these concurrent database actions simply required encapsulating database writes in a Java `Runnable` thread instance. This runs the database actions in a separate thread and the execution of responding to the user's request can be completed before the action is completed.

Security

As this project requires running user's code on the server the system is deployed on, there are a number of security concerns. Users could perform denial-of-service or other security

attacks on the system, whether intentionally or otherwise. Users could attempt to write, read, or delete files from the deployment server. This could result in the server breaking, sensitive files on the system being read, or files and directories being deleted.

These security concerns are addressed through the use of a Java Security Policy. This is a file which acts as a form of capability list. A small number of permissions are granted to the process, and if the code attempts to use other privileges, it will be prevented. The main permissions granted include access to the code files and Java environment variables, such as the user directory and class path. Other permissions granted include runtime and property permissions which are required for the Junit testing library.

Orchestration

There are a number of scenarios which could break the system due to long running processes. For example, a user could write an infinite loop which would create an infinite testing process, thus breaking the system. Another example is that database connections could break and become stuck open, forever holding a connection while the database has a finite number of connections allowed. To prevent long running processes from the users' code, including infinite loops, a timeout has been attached to each testing process spawned. If the tests do not complete within 30 seconds, the process is killed and an exception occurs. This prevents any testing processes from using system resources for an extended period of time. Likewise, a connect timeout has been added to the database connections. If any database connection is not closed within 60 seconds, the connection is automatically killed to prevent the database's connections buffer from becoming full.

4.4 Web Interface

The web interface is the interface through which the user interacts with the system. AngularJS has been used to create the web interface and was chosen over other alternatives such as React and vanilla JavaScript due to its stability as a framework and functionality over no framework (vanilla JavaScript). The interface has three main pages, is component-based and uses a service-oriented architecture. Service oriented architectures separate components from data stored in the web interface. Typically services will be injected into components, which will then call functions in the services to receive the data they require to display or use. The first page is a simple declaration, it provides the user with information about the user study and requires the user to click an `I Agree` button to continue. The second page is a form where the user can enter details about their programming experience and relevant qualifications for the study. The third page is where the user completes coding challenges. The three pages are shown in Figures 4.2, 4.3, and Figure A.1 in the appendix.

4.4.1 Components and Services

The third page (Figure 4.3) utilises the component-based and service-oriented design. Two main components comprise the page. The first component displays instructions for each coding challenge, and the second is a code editor and mock-terminal which displays compilation errors. Additionally, there are *back* and *forward* buttons to allow the user to move between challenges. Each challenge has a different set of instructions and starter code associated with it. This requires the instructions and editor components to change values displayed when a user changes to a new challenge. All the information required for the challenges are stored in a service. Abstracting the content from the components reduces how frequently the components are rendered. The abstraction also reduces the time the components take to render, as they are significantly *thinner*, with minimal data stored in them.

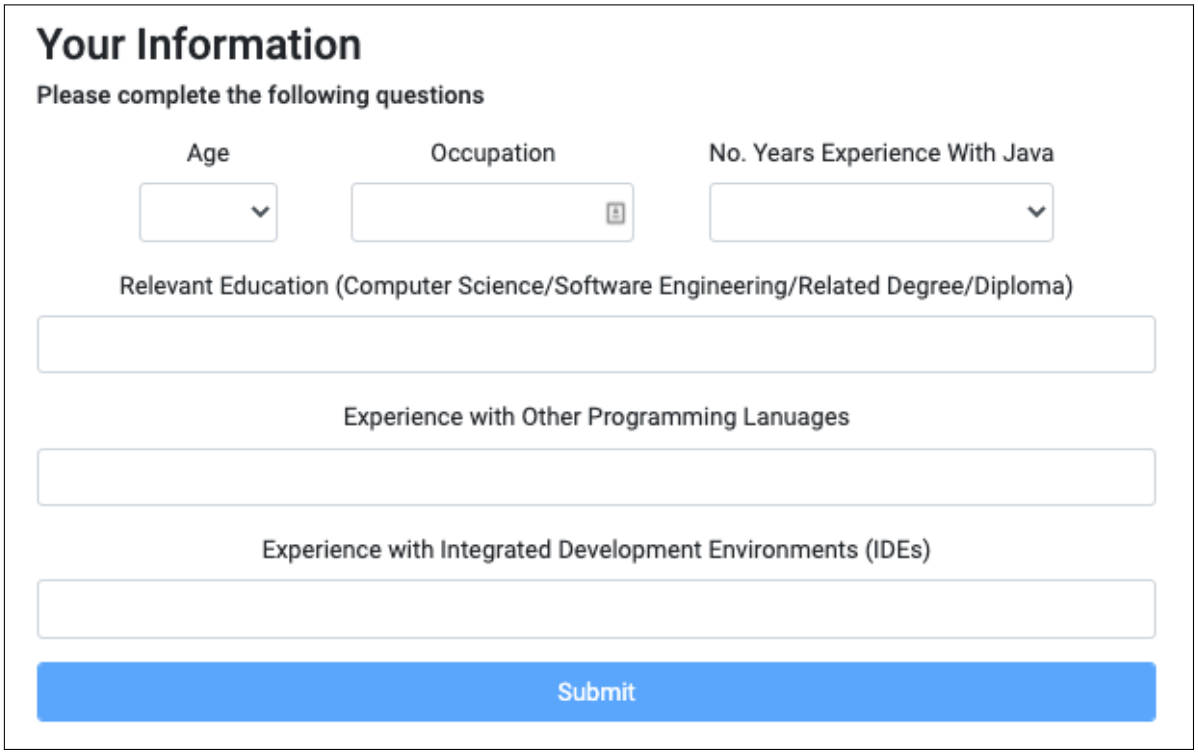


Figure 4.2: Web Interface Page 2

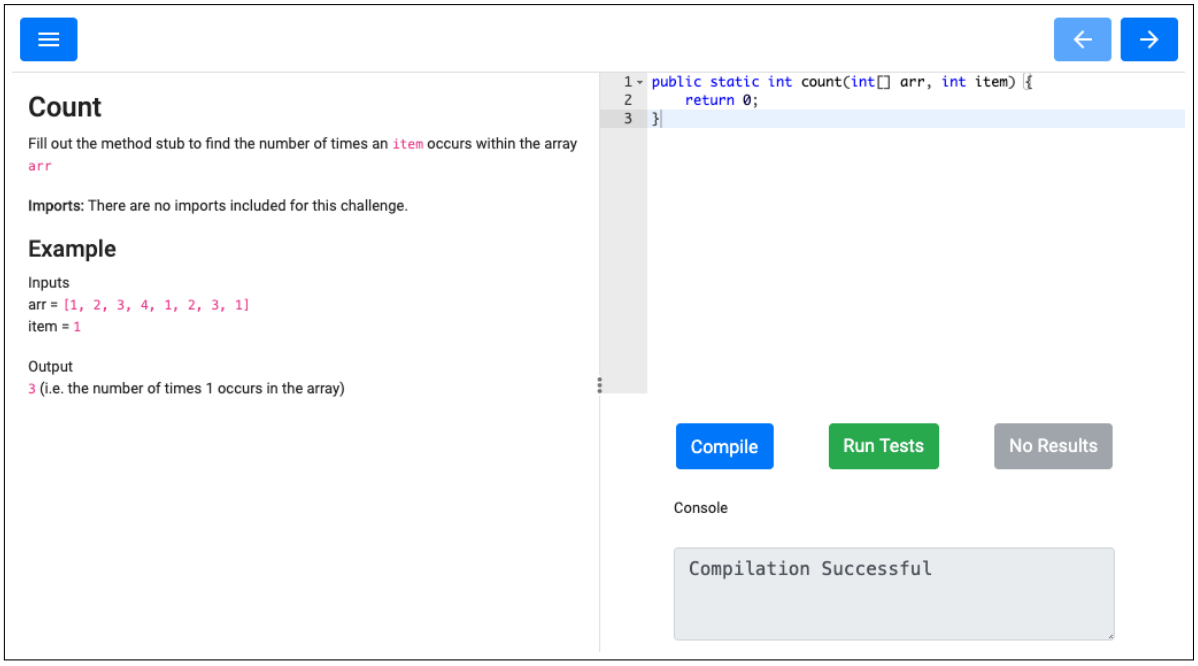


Figure 4.3: Web Interface Page 3

4.4.2 Observer Pattern

Communication between the components and challenge service on the editor page occurs through the *observer* pattern. The editor and instructions components are *subscribed* to the challenges service. When the user navigates to the next or previous challenge, the service is notified and changes the current challenge. After the current challenge has been changed, the service notifies its subscribers (the editor and instruction components) that there have been changes and the new challenge information. The components then update their challenge data and render the new challenge. Listing 4.10 shows the implementation for subscribing to the challenges service. A callback function is defined: `res => { ... }`, which updates the challenge when the challenge is changed. Additionally, in the challenge service, when the next challenge is requested the service calls `challenge.next(...)` and passes the next challenge. The `next` function informs the challenge to perform the callback on all subscribed objects.

```
this.challenges = this.challengesService.getCurrentChallenge();
this.s = this.challenges.subscribe(res => {
  this.challenge = res;
  this.code = res.starterCode;
});
this.challengesService.initChallenges();
```

Listing 4.10: Subscription to Challenges Service

4.4.3 Auth Guards

Due to the foreign key constraints in the database discussed in Section 3.2.3, where compilations and test results cannot exist without a `user_id`, information must be added to the database in a particular order. Additionally, as the application is being used for a user study, users must accept a declaration before they can proceed with the study. As every table in the database requires a user id to be present, users are prevented from accessing the challenges and editor unless they have both accepted the declaration (which generates a session key to be used as the `user_id`) and completed the information form (which adds the user id and information to the `user_information` table in the database).

To prevent users from accessing the challenges, or entering details before they have accepted the declaration, two `Auth Guards` are used. These guards implement the `CanActivate` interface from Angular Routing (an in-built component routing system within AngularJS), which requires a `canActivate` function. The function returns a boolean: `true` if the user can proceed and `false` if they cannot. The first guard prevents users from proceeding past the declaration page unless they have accepted the terms of the user study. This guard uses a service which handles session information and returns true if a session key exists. A session key is only generated once a user has accepted the declaration. The second guard both checks that the user has accepted the declaration and that they have completed the information form on the second page of the web interface.

4.4.4 Code Editor

The code editor used is an Angular component for the Ace editor [28]. Ace is a widely used, JavaScript embeddable code editor which can be used in any web page. The Amazon Cloud9 online IDE uses Ace as the primary editor. The component provides handles

as HTML/Angular attributes for a number of configurations, including updating the code and the style of the editor. Listing 4.11 shows how the editor is embedded into the web interface. The `config` attribute is set to a configuration JSON object holding the language and syntax highlighting information. The `value` attribute is a two-way binding. The `code` variable is a string which holds the code in the editor. When the code is either updated by the challenges service or by the user in the editor, the editor and variable are updated, respectively.

```
<ace [config]="config" [(value)]="code" id="editor"></ace>
```

Listing 4.11: Embedded Ace Editor

4.5 Database

The database for the application is implemented using PostgreSQL. The architecture of the database has been described in Section 3.2.3.

Connecting to the database is managed using the JDBC library and prepared statements. JDBC uses a connection string specifying the server, port and database name to abstract the connection to the database. Prepared statements used help to manage the security of the database. These statements pre-compile the SQL statements and then add the values into the statements after. As user entered data and code is added to the database, ensuring users do not perform an SQL injection attack is crucial. Prepared statements also convert all data added into a query to a string before adding them, thus preventing an injection attack. Listing 4.12 shows how a prepared statement is implemented in Java. The SQL query is created with question marks (?) in place of values, which then compiles the statement so that data can be added to the query after. This also increases the speed of the query as it does not require compiling every time it is used. After the query is prepared, the data is set using the `setString(...)` method calls, and then the query is executed.

```
db = DriverManager.getConnection(Main.DATABASE_CONN_STRING,
   Main.DATABASE_PROPERTIES);
PreparedStatement stmt = db.prepareStatement("INSERT INTO
   user_information" +
            "(id, age, occupation, java_experience, education,
               other_langs, ide_experience, magic_number) " +
            "VALUES (?, ?, ?, ?, ?, ?, ?, ?)");
stmt.setString(1, id);
stmt.setString(2, age);
stmt.setString(3, occupation);
stmt.setString(4, java_experience);
stmt.setString(5, education);
stmt.setString(6, other_langs);
stmt.setString(7, ide_experience);
stmt.setString(8, magic_number);
stmt.executeUpdate();
```

Listing 4.12: Prepared Statement Example

Chapter 5

Evaluation and Results

A number of methods of evaluation have been used throughout this project. The web interface has been evaluated through the use of software tests, the web server through the use of logging, and the overall system has been checked for robustness and ease of use through a pilot study before completing a full user study. We now examine in more detail and present the overall results from the study.

5.1 Web Interface Test Suite

The Angular web framework provides a testing framework in the standard install and automatically generates some tests on the creation of components and services. This framework is called Karma, and the tests generated simply test if a component or service is successfully created. By default a number of these tests will fail as components and services may have dependencies to other services which will not be injected in the test suite. Fixing these tests involved configuring the test bed to inject required services and modules. Additionally, a number of tests have been added to ensure that the main logic of the web interface code is tested. Overall, there are 55 tests in the test suite, which aim to ensure the correctness of the code. These tests also aim to prevent any future changes to the codebase from breaking existing functionality.

Figure 5.1 shows the coverage of the test suite when performed on the web interface. Overall statement coverage is at 95.97%, with branch coverage at 88.16%. There are a few cases which could be added in the *editor* component to increase these numbers, but overall the coverage is good. The low branch coverage in the *declaration* component include safety checks for handling identifiers assigned to users. The two branches not covered are the `?.id` and `mag || 0` checks. These checks exist as users can navigate to the application with or without an identifier in the url. In the case that there is no identifier in the url, the `mag` variable is set to 0 to handle this. Adding a test case for this would be trivial, however it does not affect the execution of the interface for the user so it is not a crucial test.

```
const mag = this.route.snapshot.queryParams?.id;
localStorage.setItem('mag', mag || 0);
```

Listing 5.1: Declaration Missing Branches

The main cases in the *editor* component which are not covered include handling of a time interval for auto-compilation, and sending DELETE requests to the `storeUser` endpoint. Both of these cases relate to non-required features which will not affect the application significantly if they were to break. While this could eventually affect the quality of the data

All files

95.97% Statements `238/248` **88.16%** Branches `67/76` **97.01%** Functions `65/67` **95.59%** Lines `217/227`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
src	100%	3/3	100%	0/0	100%	0/0	100%	3/3
src/app	100%	3/3	100%	0/0	100%	1/1	100%	2/2
src/app/components/challenge-details	100%	8/8	100%	0/0	100%	3/3	100%	7/7
src/app/components/challenge-editor	100%	2/2	100%	0/0	100%	2/2	100%	1/1
src/app/components/declaration	100%	15/15	75%	6/8	100%	4/4	100%	13/13
src/app/components/editor	89.69%	87/97	86.96%	40/46	88.24%	15/17	88.89%	80/90
src/app/components/editor-header	100%	9/9	100%	0/0	100%	6/6	100%	8/8
src/app/components/information-form	100%	9/9	100%	0/0	100%	3/3	100%	8/8
src/app/components/menu-modal	100%	18/18	100%	0/0	100%	5/5	100%	17/17
src/app/services	100%	83/83	95.45%	21/22	100%	26/26	100%	77/77
src/environments	100%	1/1	100%	0/0	100%	0/0	100%	1/1

Figure 5.1: Web Interface Test Suite Coverage

produced, or the storage on the web server, these are not critical and can be fixed in the case that they do break.

The test suite has been incorporated into a continuous integration (CI) pipeline on GitLab. This pipeline ensures that when any code is pushed to the repository, the test suite is run. If any of the tests fail in the CI pipeline, the branch the code has been checked into is unable to be merged to the master branch. Additionally, a notification is sent to the developer who introduced the error in the code that the pipeline has failed. Over the duration of the project there has been 83 CI pipelines run on the web interface. Of these 83 pipelines, there have been 68 successful pipelines and 14 failed. Of the failed pipelines, a number were due to errors being introduced to the code. A small number of the pipelines failed due to tests becoming outdated. These pipelines required the test suite to be updated, ensuring the tests would still be useful for future pipelines. Overall, the test suite and CI pipeline has been shown to prevent bugs from being added to the master branch and deployed into the final application available for the user study.

5.2 Logging

The quality of the web server is assessed through the use of logging. Rather than writing tests for the server, which could be an arduous process due to the number of files written and processes created, logging offers insight into any errors which occur through the execution of the server. A logger has been added to the server which allows logging of messages throughout the execution. The server logs actions such as compile and testing requests, adding and deleting users, and adding information to the database. Additionally, the logger also logs the stack trace of any errors which occur throughout execution of the program.

Between 1st September and 10th October 2020, a log file with 6924 lines has been generated. Of these 6924 lines, 210 are errors from the server. This is an error rate of approximately 3%, which initially appears to be high. However, upon closer analysis of these error messages, they all appear to be the same type. The errors raised throughout execution occur due to web connections to the server being ended before the server expects. These errors are fairly innocuous and in no way affect the execution of the server. As the server has been running since the 1st September and has not reported any major errors, no users have reported any issues with the application, and the server has never crashed, we can conclude that the server is significantly robust for the purpose of the user study.

5.3 Pilot Study

Prior to the main user study, a number of users took part in a pilot study. The aim of this pilot study was to evaluate the application to ensure it was fit for use in the main user study. The pilot study resulted in three main discoveries about the system.

5.3.1 Method

This pilot study was conducted with the developed web application deployed on a server at the School of Engineering and Computer Science at Victoria University of Wellington. The server used runs Arch Linux, version 5.7.7 and the Java version used is OpenJDK Java 15. A group of four students from the Engineering Project (ENGR489) course at the University were selected to participate and evaluate the application. For these students, the author observed the experimental sessions, allowing the students to ask questions. Additionally, their use of the application was monitored to ensure they used it correctly, and to discover

any issues with the user experience of the application which could be changed before the main user study.

5.3.2 Findings

Firstly, the time for completing the challenges exceeded the expected amount of time. We expected the challenges to take somewhere between 30 and 45 minutes to complete. Participants found that the challenges could take up to two hours to complete. To ensure we were not asking too much of our participants, we amended the instructions for the main user study. Instead of stating *'the challenges should take approximately 30 to 35 minutes'*, we now state *'you are only required to spend 30 to 45 minutes participating.'* This amendment ensured our users knew our expectations for the study, while also allowing them to spend longer on the challenges if they wished.

Secondly, we found that some of the constraints on the database were too restrictive for participants to enter their information in the information form on the second page of the web interface. For example, there was a restriction on the number of characters a participantscould have for their occupation. Initially, this field was set to 15 characters. This caused an issue when a participant tried to enter *'Software Engineering Student'* as this was more than 15 characters. Another issue raised because of this was that participants could enter more characters in the web interface than are allowed in the database. To fix the character limit, the length of the occupation field was increased to 40 characters. This provided a suitable fix to allow the participant who raised this issue to enter their information. If more participants had brought up similar issues, a fix which could have been implemented would have been to add validation to the user information form on the web interface. This validation would ensure that participants could not enter more characters into any field than are allowed in the database. This would prevent cases where a participant enters too much information and an error occurs when trying to add the information to the database. The web interface currently handles these errors to ensure it doesn't break, however the interface doesn't proceed from the page with the form and does not inform the participant of the error. As no participants since the pilot study have had any issues with this, we conclude that increasing the character limit on the occupation field was sufficient to resolve this issue.

Finally, the pilot study resulted in a less than ideal amount of data being generated. We were concerned that not enough edits would be generated as a part of the main user study. Additionally, we were also concerned about participants using third-party tools such as Eclipse or IntelliJ to complete the challenges, and pasting their results back into the web application. This would also result in minimal data being generated. As a result of these, we updated the web interface to add an automatic compilation feature. This feature sends a compile request to the web server when a participant stops writing code for five seconds. Automatic compilation is a similar feature to the continuous building feature included in most IDEs. For example, the Eclipse IDE allows users to configure the time interval between writing code and automatic compilation. This results in both more data being generated, and participants being provided more feedback on the code they are writing, as they usually would in a standard development session.

5.4 User Study

Prior to the user study, Human Ethic approval was obtained. The Victoria University of Wellington Human Ethics Committee approved this user study under the application number 0000028477.

5.4.1 Method

The main user study followed a similar method to the pilot study, with a few key differences. There was no observation of users throughout the main study. Users were sent a URL to the web application so they could access and complete the coding challenges in their own time. A total of 19 users participated in the study, with ten of these sourced by word of mouth or internal mailing lists and supplied vouchers as required by the Human Ethics Committee. The other nine participants were sourced through social media: LinkedIn and Twitter. Participants sourced via social media were not offered vouchers, as it would have been too difficult to contact and distribute the vouchers. Ten of the participants identified themselves as students, with the other nine identifying as developers or engineers. 11 of the participants had one to three years of Java experience, five had between 4 and 6, with three having greater than seven years Java experience.

5.4.2 Results

There are a number of interesting results from the user study. This section will discuss and explain some of these results.

Data Generated

A total of 1151 edits, 192 test requests, and 5503 pieces of instrumentation data have been generated as a part of the main study. Table 5.1 shows the breakdown of compilations per challenge in the study. The programmatically larger and more difficult challenges such as *Flatten*, *Covariance*, and *Permutations* have the largest number of compilations. Similarly, the smaller challenges such as *Max 3* have the least number of compilations. This is to be expected, as the more difficult a challenge, the more time a user will spend on it, which will then result in a larger number of compilations.

Table 5.1: Number of Compilations Collected per Challenge

Challenge	Number of Compilations
Count	82
Covariance	184
Exception Handling	73
Exception Handling 2	131
Flatten	169
Generics	57
Generics 2	107
Max 3	33
Permutations	123
Reverse	48
Search	73
Sort	71

Overall Stages

Figure 5.2 shows a representation of the proportion of edits which reach each stage of compilation. Code edits are shown on the y-axis, with compilation stages being represented along the x-axis. The stages shown are in the compiler pipeline order described in Section 2.2. Additionally, the *attribution* stage has been split into *Attribution (Name Resolution)* and *Attribution (Type Checking)*.

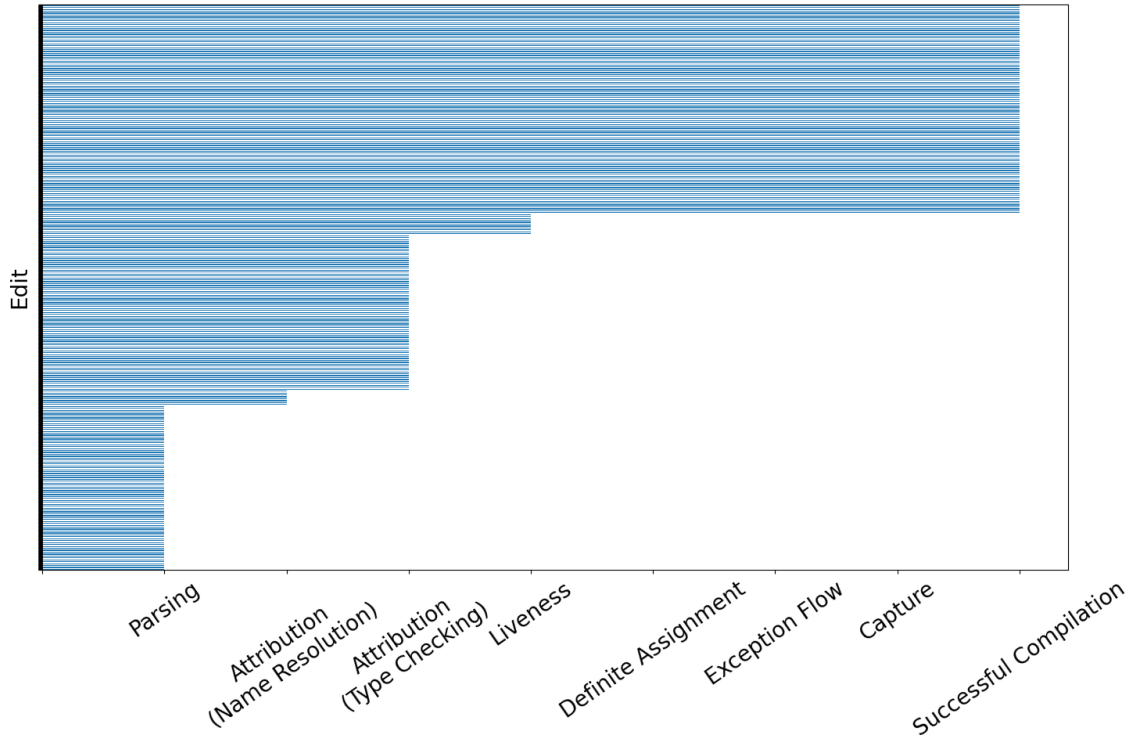


Figure 5.2: Stages reached by compilation across all participants.

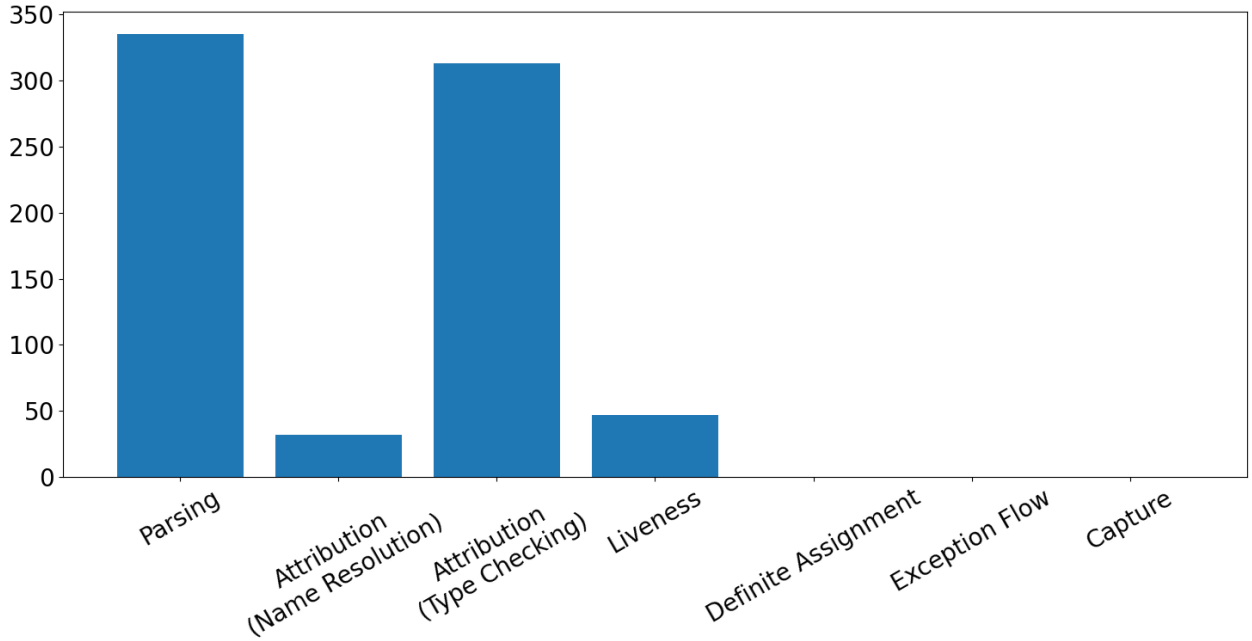


Figure 5.3: Illustrating number of edits failing at each stage.

The main band of edits across the top of Figure 5.2 show successful compilations, with all other bands showing failed compilations and the respective stage at which they failed. While a slight majority of compilations are shown to fail (63.2%), there is a significant number of compilations which succeed. We can see that, of the compilations which fail, the majority are in the *parsing* (46%) and *type checking* (43%) stages. Overall, the number of failures decreases, with more errors occurring in the earlier stages of the compiler. This is further illustrated in Figure 5.3. Most of the compilation errors occurred in *parsing* and *type checking*, with only a few in *name resolution* and *liveness*. No compilation errors arose during *definite assignment*, *exception flow* or *capture*.

Parsing errors tend to be users missing semicolons at the end of lines in their code, accidentally using the wrong syntax, or typos. Typos also contribute to failures in name resolution, as a user may incorrectly type a variable or method name, resulting in a failed compilation. Overall, it appears that the majority of errors come from users compiling code with typos, missing syntax or incorrect types. An interesting observation here is that *name resolution* doesn't trigger often. One might expect failures to reduce as a compilation passes further through the pipeline. However, this data shows that for *name resolution*, this is not the case.

Stages in Specific Challenges

There are some interesting observations about compilation errors in different challenges. Most of the challenges follow a similar distribution as the overall errors shown in Figure 5.3. Figure 5.4 shows the results for the *permutations* challenge to illustrate one example. The most errors by far occurred in the *parsing* stage of compilation, followed by *type checking*. This shows a very similar distribution to the overall population, which is to be expected.

Perhaps a more interesting observation is the distribution of errors for the *generics* challenges. Figures 5.5 and 5.6 show the distributions for the *generics* and *generics 2* challenges. In these challenges, the user is asked to use the Java language's complex generic type system. This system allows users to specify objects and methods which have a generic type. This generic type can later be specified as any other subtype of the declared upper bound, which is `java.lang.Object` by default. In these challenges, the stage which has the most errors is *type checking* as opposed to *parsing* for the overall population. This result is likely due to the nature of these two challenges testing one of the most complex aspects of the Java language. A possible scenario is that users may not be extremely familiar with the generic type system in Java. Another possibility is that the code to pass these challenges is not extremely complex for the *parsing* stage, but the type system is. However, the large number of parsing errors could also occur as a result of the syntax for declaring generic classes. Listing 5.2 shows an incorrect placement of a generic declaration which a user attempted to compile during the user study. This error would not be classified as a *type checking* error even though the error arises because of misuse of the type system. The error would, in fact, be classified as a *parsing* error. A higher number of parsing errors could be reported when errors are occurring because of this kind of type system misuse.

```
class <T> Box { ... }
```

Listing 5.2: Incorrect Generic Class Declaration

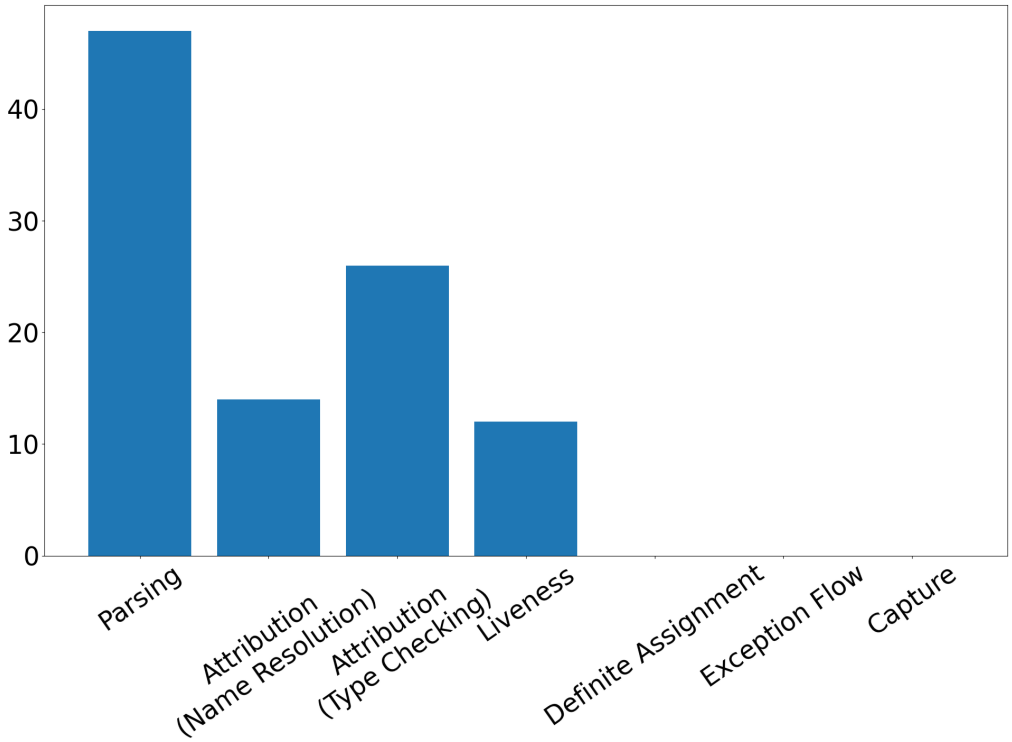


Figure 5.4: Stages failed for the *permutations* challenge.

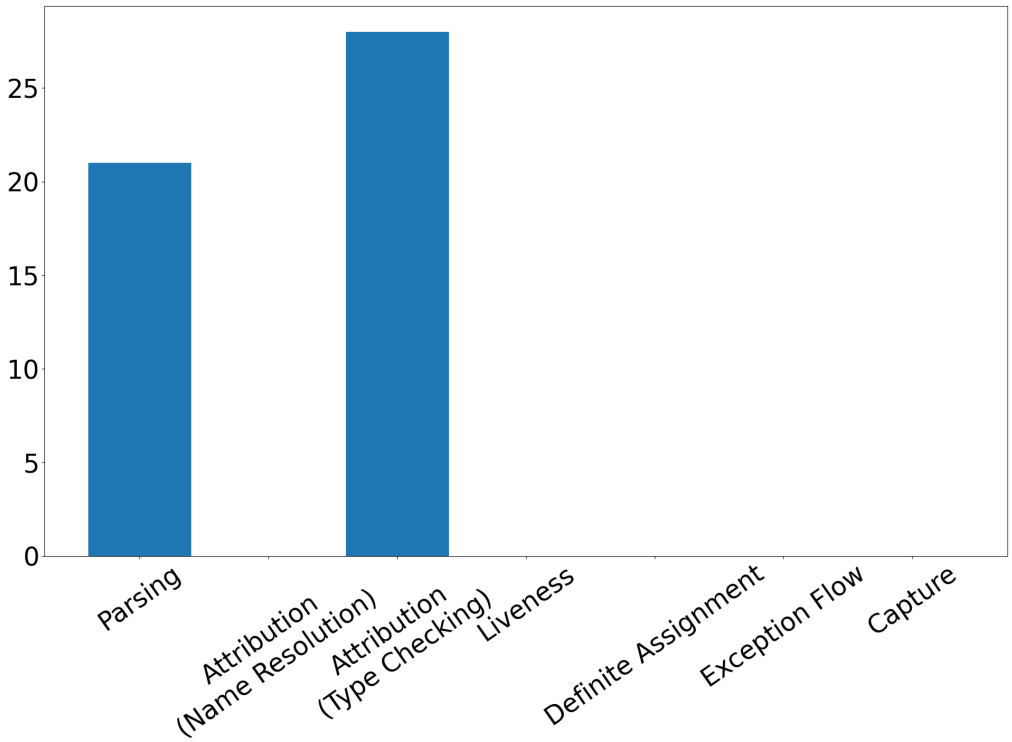


Figure 5.5: Stages failed for the *generics* challenge.

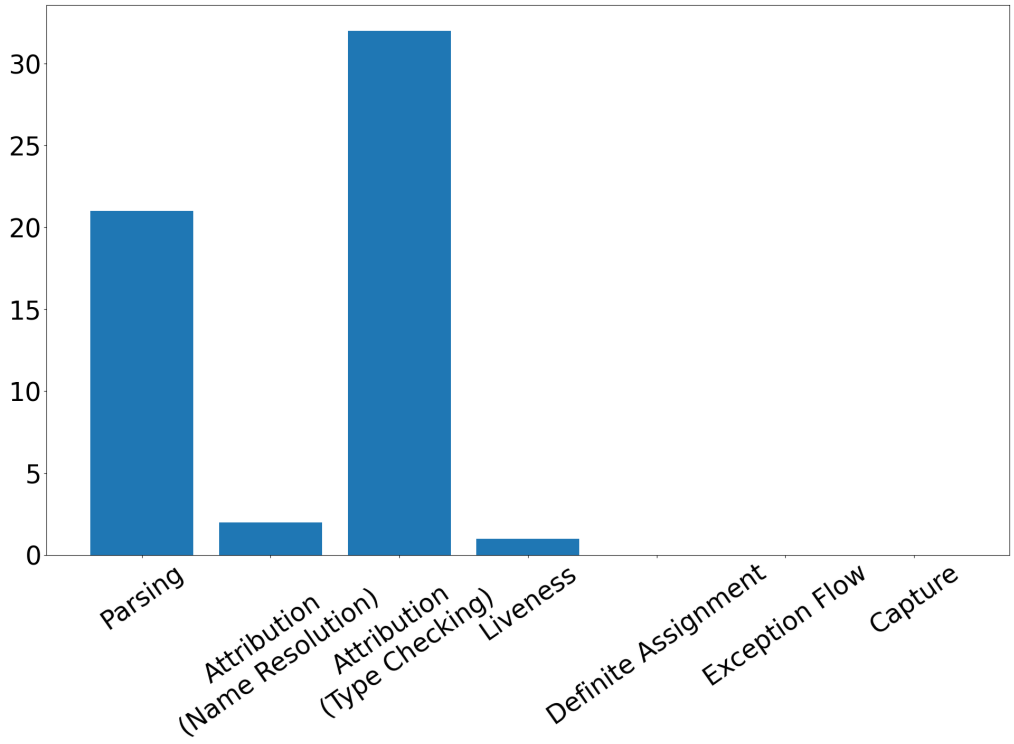


Figure 5.6: Stages failed for the *generics 2* challenge.

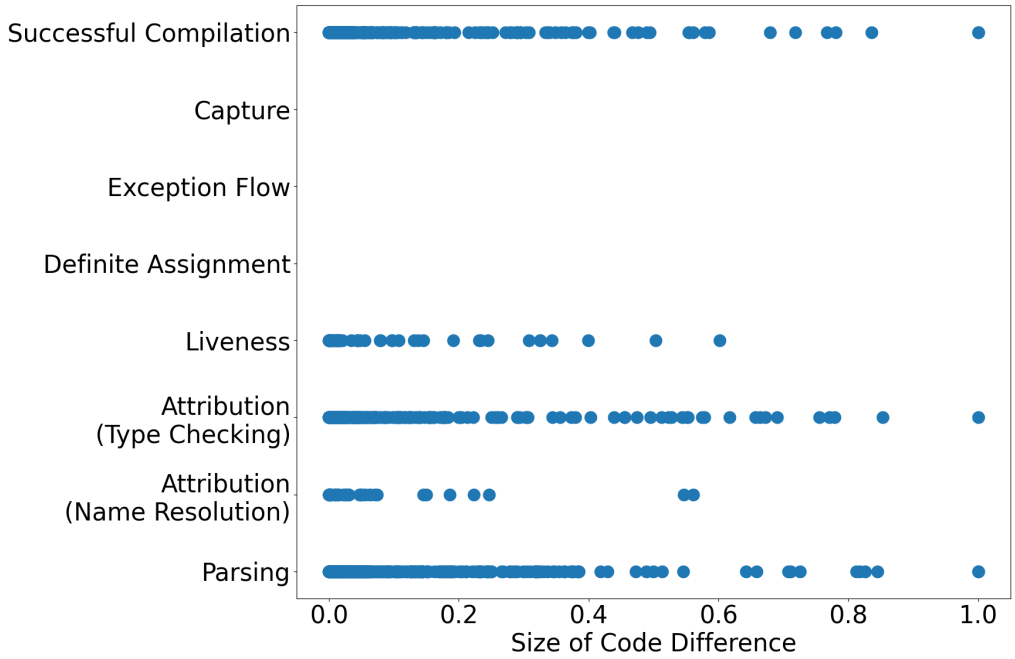


Figure 5.7: Result of compilation by size of code edit.

Size of Code Edits

Another metric we explored was to see how failures in compilations relate to the size of an edit to a user's code. Figure 5.7 shows the compilation results by the size of the difference between two edits. To compute this analysis, we generated the size of an edit through a code diff. The Python 3 `difflib` library and `SequenceMatcher` object were used for this. Code differences in `SequenceMatcher` are generated following the formula: $1.0 - \frac{2.0 \times M}{T}$ where T is the total number of characters in the code from two edits, and M is the number of matches between edits. The result is a number between 0.0 and 1.0, where 0.0 is no changes to the code and 1.0 is when the code is entirely changed. We see that the edits are grouped towards 0.0, with a tail towards 1.0. This shows that the majority of edits tend to have fewer changes. We note there is a more consistent spread across the size of edits for *parsing*, *type checking*, and *successful compilations* than *liveness* and *name resolution*. This data could potentially show some correlation between edit size and these compilation stages, however the current data set is likely too small to draw any conclusions here.

5.4.3 Threats to Validity

There are a number of threats to the validity of this study. A few of these are discussed below:

Challenges

As previously discussed, the challenges created for this project are limited and are likely not to fully replicate a standard development session. While the challenges were used as a feasible way to produce this data in the timeframe for the project, completing a user study in a development workplace or with larger projects could produce different results.

Automatic Compilation

The automatic compilation feature added to the web application could affect the validity of the instrumentation produced from the compiler. If a user stops to think while they have not completed a line of code and automatic compilation occurs, this could lead to a higher number of parsing errors. We note, however, that several modern IDEs such as Eclipse and IntelliJ also offer this feature.

Web Application Simplicity

We acknowledge the simplicity of the web application developed for this study. There are a number of features such as debuggers and code automatic completion which are not present in the web application. These features, however, are present in most modern IDEs. Because of this simplicity, developers may behave differently when they have access to these tools. Another threat due to the simplicity is the order of the coding challenges. Developers may behave differently when completing challenges in a particular order. As the challenges were presented to all participants in the same order, there could be different results for a randomised set of challenges.

User Selection

The number of participants in this study was quite low and the majority were related to the University. This could cause some hidden trends in the data due to the users having similar programming backgrounds. Additionally, a study with a larger number of participants is likely to be more representative, could result in more data being generated and different conclusions.

Chapter 6

Conclusions and Future Work

6.1 Future Work

While the application and user study have been successful for this project, there are a number of areas for improvement and future work.

User Study

There are several places where the user study could be improved. For example, randomising the order of challenges the participant receives could ensure that participants complete more of the challenges. Additionally, participants may behave differently when they complete challenges in different orders. Another user study could be undertaken to generate more data and compare with the findings of this report. Furthermore, analysis of the data generated in this project has been constrained by time and therefore more in-depth analyses and conclusions could be drawn about the data. In future studies, a larger number of participants could be involved. This would result in more diversity and a more representative sample of the developer population, allowing for better conclusions to be drawn. Additionally, more instrumentation could be added to Javac and more variety of data generated. For example, full logs of compilation errors could be saved and analysed to discover the main causes of errors in compilation stages. An example of this sort of analysis could be discovering what percentage of *parsing* errors occur because a user misses a semicolon.

Using the Findings

As discussed in Section 1.3, the findings from this report can be used to improve the design of incremental and distributed compilers. Experiments in using these findings for compiler design is a logical next step. For example, as we have shown, the majority of errors in compilation occur in the *parsing* and *type checking* phases of compilation. A distributed compiler could use these findings to inform how the compiler is separated between a web interface and web server. *Parsing* and *type checking* could all be moved into a web interface, separate from the web server and performed on the user's computer when they compile a program. The web interface could then pass the generated and attributed AST to the web server for the final checks and code generation, thus improving the performance of the compiler and reducing the network workloads.

Web Application Improvements

The web interface test suite can be improved and extended to more comprehensively cover the code. While the current test suite covers a significant amount of the code in the web interface, there is always room for improvement. High branch coverage and further thought into cases which could arise throughout use of the application could improve the test suite. The error logs discussed in Section 5.2 could be improved. The issue where the server is not expecting connections to be ended when they are could be fixed to improve the overall logging in the application. Removing these error logs would allow logs of more critical errors to be easily found and addressed in the case they do arise.

6.2 Conclusion

This project has focused on generating realistic compiler workloads through the use of a web application and user study. This user study involved 19 participants from both educational and professional backgrounds. The web application developed allows users to complete a number of coding challenges while compiling and testing their code. A range of coding challenges have provided a variety of scenarios which a developer may experience in a standard workplace and cover both algorithmic challenges and language feature use. Throughout development of the web application, a testing suite and CI pipeline ensured that bugs have not been introduced to the codebase. Similarly, during the user study, logging has been used to ensure that no critical errors have occurred in the web server. Finally, a number of interesting results have been discovered across 1151 compilations and 5503 lines of instrumentation generated from the compiler. Overall, the size of a code edit appears to affect *name resolution* and *liveness* but not other stages such as *parsing* and *type checking*. Furthermore, *parsing* and *type checking* are the most frequent stages of compilation failed for the majority of challenges. However, for those challenges involving more complex type systems, such as generics, *type checking* becomes a more prominent failure for compilation.

Bibliography

[1] E. C. Berkeley, "Counting holes: Punch-card calculating machines," in *Giant brains; or, Machines that think*. Wiley, NY, 1949, pp. 42–65.

[2] E. G. Nilges, "A brief history of compiler technology," in *Build Your Own .NET Language and Compiler*. Berkeley, CA: Apress, 2004, pp. 1–13. [Online]. Available: https://doi.org/10.1007/978-1-4302-0698-9_1

[3] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[4] International Business Machines Corp., "Eclipse platform technical overview," Eclipse, Tech. Rep., 2006, available at eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[5] M. A. Hammer, J. Dunfield, K. Headley, M. Narasimhamurthy, and D. J. Economou, "Fungi: Typed incremental computation with names," *CoRR*, vol. abs/1808.07826, 2018.

[6] R. Mecklenburg, *Managing Projects with GNU Make, 3rd Edition*. O'Reilly Media, Inc., 2004.

[7] J. Popple, "Incremental compilation and its implementation in the PECAN programming environment generator," B.A. (hons) thesis, Australian National University, Nov. 1987.

[8] M. Woerister. Incremental Compilation. rust-lang.org. (accessed May. 24, 2020). [Online]. Available: https://blog.rust-lang.org/2016/09/08/incremental.html

[9] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students." Association for Computing Machinery, 2003, p. 153–156.

[10] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, 2014, p. 724–734.

[11] D. McCall and M. Kölling, "A new look at novice programmer errors," *ACM Transactions on Computing Education*, vol. 19, no. 4, Jul. 2019.

[12] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, 2014, pp. 1–8.

[13] Y. Liang and L. Yansheng, "An incremental compilation algorithm for the java programming language," in *2012 7th International Conference on Computer Science Education (ICCSE)*, 2012, pp. 1121–1124.

[14] T. Lindhold, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java Virtual Machine Specification*, Oracle Corporation Std., Feb. 2020.

[15] M. A. Ertl and A. Krall, "Instruction scheduling for complex pipelines," in *Compiler Construction*, U. Kastens and P. Pfahler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 207–218.

[16] Rust Lang. "Incremental Compilation". rust-lang.org. (accessed Oct. 13, 2020). [Online]. Available: https://doc.rust-lang.org/edition-guide/rust-2018/the-compiler/incremental-compilation-for-faster-compiles.html

[17] Oracle Corporation. "Compilation Overview". openjdk.java.net. (accessed May. 24, 2020). [Online]. Available: https://openjdk.java.net/groups/compiler/doc/compilation-overview

[18] ——. "Annotations Basics". oracle.com. (accessed May. 31, 2020). [Online]. Available: https://docs.oracle.com/javase/tutorial/java/annotations/basics.html

[19] "Variable scoping and definite assignment," in *A Programmer's Introduction to C# 2.0*, E. Gunnerson and N. Wienholt, Eds. Berkeley, CA: Apress, 2005, pp. 113–117.

[20] N. Matsakis, *Responsive Compilers*, 2020. [Online]. Available: https://www.youtube.com/watch?v=N6b44kMS6OM

[21] G. Maudoux and K. Mens, "Bringing incremental builds to continuous integration," in *Proc. 10th Seminar Series Advanced Techniques & Tools for Software Evolution*, 2017, pp. 1–6.

[22] M. Shal. What is tup? gittup. (accessed Sep. 18, 2020). [Online]. Available: http://gittup.org/tup/

[23] NixOS.org. Reproducible builds and deployments. NixOS.org. (accessed Sep. 18, 2020). [Online]. Available: https://nixos.org/

[24] T. Cooper and M. Wise, "Achieving incremental compilation through fine-grained builds," *Software: Practice and Experience*, vol. 27, no. 5, pp. 497–517, 1997.

[25] A. Celik, A. Knaust, A. Milicevic, and M. Gligoric, "Build system with lazy retrieval for java projects," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 2016, p. 643–654.

[26] P. W. Sathyanathan, W. He, and T. H. Tzen, "Incremental whole program optimization and compilation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 221–232.

[27] S. P. Reiss, "An approach to incremental compilation," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '84. Association for Computing Machinery, 1984, p. 144–156.

[28] ace.c9.io. "Ace: The High Performance Code Editor For The Web". ace.c9.io. (accessed Oct. 13, 2020). [Online]. Available: https://ace.c9.io/

Appendix A

Figures

Incremental Compilation Workloads - ENGR489 Project

Information for Participants

You are invited to take part in this research. Please read this information before deciding whether or not to take part. If you decide to participate, thank you. If you decide not to participate, thank you for considering this request.

Who am I?

My name is Philip Oliver and I am an ENGR489 Honours student in Software Engineering Victoria University of Wellington. This research project is work towards my Honours report.

What is the Aim of this Project?

This project aims to generate some realistic compilation workloads and to explore how these workloads affect the compiler. Your participation will support this research by providing valuable insight into compilation sessions which will aid compiler design, in particular for incremental compilers. This research has been approved by the Victoria University of Wellington Human Ethics Committee (application no. #0000028477).

How Can You Help?

You have been invited to participate because you have experience programming in the Java programming language. If you agree to take part, you will complete a short survey followed by a series of programming exercises using a simple online IDE. The survey will ask you questions about your programming experience and then ask you to complete some programming challenges. The survey will take you approximately 30-45 minutes to complete. During your completion of the programming challenges you may be asked to share your screen on Zoom. This will be done so we can identify any issues with the application while you are participating. No audio or video will be recorded over Zoom. A voucher worth $10 will be offered to participants.

What Happens to the Information You Give

This research is anonymous. This means that nobody, including the researchers will be aware of your identity. By answering it, you are giving consent for us to use your responses in this research. Your answers will remain completely anonymous and unidentifiable. Once you submit the survey, it will be impossible to retract your answer. Please do not include any personal identifiable information in your responses.
Personal details will be collected only for those who wish to enter the prize draw/request a copy of the final report. All personal details will be received separately from the survey data and will be held in confidence. This ensures that your answers to the survey questions will not be linked to your identity. All identifiable data will be destroyed on 7 November 2020.

What Will This Project Produce?

The information from my research will be used in my Honours report and/or academic publications and conferences.

If You Have Questions, Who Can You Contact?

Any questions should be directed to either - emails have already been provided:

Student	**Supervisor**
Name: Philip Oliver	**Name:** David J. Pearce

Disclaimer

By continuing, you agree to the above information and are giving consent for us to use your responses in this research.

☐ I Agree

Continue

Figure A.1: Web Interface Page 1