

# Automated Mutation Testing for Concurrent Software

Dept. of Computer and Information Sciences  
University of Strathclyde

Patrick Gray

August 2019

This dissertation was submitted in part fulfilment of requirements for  
the degree of MSc Software Development

## DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc Software Development of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

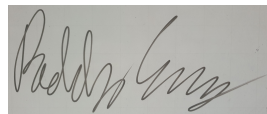
I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [☒] No [☐]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is **15,020**

I confirm that I wish this to be assessed as a Type 5 Dissertation

Signature:

A handwritten signature in black ink, appearing to read 'Paddy Gung', is written over a light gray rectangular background.

Date: 19/08/2019

## Abstract

Concurrency allows for multiple computations to be executed within a program at once (Mois 2015). Programmers utilise this feature for the benefits in performance, however, execution of these computations is inherently unpredictable and can lead to a variety of problems. Farchi et al. (2003) define 'bug patterns' as the implementation of common errors that programmers encounter when using concurrency. Bradbury et al. (2006) build on this concept by introducing a variety of 'mutation operators' that can be applied to concurrent features in Java. A mutation operator defines a mechanism that alters a segment of code for the purpose of evaluating the effectiveness of a system's testing suite. If after applying a mutation to a code segment, the tests proceed to fail, then the tests have successfully identified a change in behaviour of the system; the mutation has been killed. However, if the tests continue to pass despite the changes in code, then tests have been unsuccessful at catching a mutation and therefore have not been designed effectively.

Having reviewed the relevant literature, the objectives of this project were to build an automatic mutation tool, apply mutations to different concurrent software systems and evaluate the effectiveness of the corresponding unit tests. The mutation tool was successfully built and was tested on two different systems: the *Banking* and *Incrementer* systems. The results of these experiments indicate that it is possible to achieve high levels of support for concurrent software with mutation scores of 86% and 100%, respectively for the *Banking* system and the *Incrementer* system. These scores represent the number of mutations that were successfully caught by unit tests compared to the total number of mutations applied to the system.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Concurrency . . . . .	6
2.1.1	Threads . . . . .	6
2.1.2	Atomic Operations . . . . .	7
2.1.3	Synchronization and Locks . . . . .	9
2.1.4	Liveness . . . . .	10
2.1.5	Executor Service . . . . .	10
2.2	Mutation Testing . . . . .	11
2.3	Regular Expressions . . . . .	13
2.4	Concurrent Bug Patterns . . . . .	14
2.4.1	Unprotected Code . . . . .	15
2.4.2	Unexpected Interleavings . . . . .	16
2.4.3	Blocking Code . . . . .	16
2.5	Concurrent Mutation Operators . . . . .	17
2.5.1	Modify Parameters of Concurrent Method . . . . .	18
2.5.2	Modify the Occurrence of Concurrency Method Calls . . . . .	18
2.5.3	Modify Keywords . . . . .	18
<b>3</b>	<b>Methodology</b>	<b>19</b>
3.1	Selected Mutation Operators . . . . .	19

3.1.1	MXT - Modify Method-X Timeout . . . . .	19
3.1.2	MSP - Modify Synchronized Block Parameter . . . . .	20
3.1.3	RTXC - Remove Thread Method-X Call . . . . .	21
3.1.4	RCXC - Remove Concurrency Mechanism Method-X Call . .	22
3.2	Mutation Tool . . . . .	23
3.3	Concurrent Software . . . . .	25
3.3.1	Banking System . . . . .	26
3.3.2	Incrementer System . . . . .	29
3.4	Unit Tests . . . . .	31
3.4.1	Banking System Concurrent Tests . . . . .	32
3.4.2	Incrementer Concurrent Tests . . . . .	34
3.5	Software Engineering Process . . . . .	35
<b>4</b>	<b>Analysis</b>	<b>37</b>
4.1	Results . . . . .	37
4.1.1	Banking System Results . . . . .	38
4.1.2	Incrementer System Results . . . . .	42
4.2	Results Analysis . . . . .	43
4.3	Methodology Analysis . . . . .	45
<b>5</b>	<b>Recommendations</b>	<b>48</b>
<b>6</b>	<b>Conclusion</b>	<b>50</b>
	<b>Bibliography</b>	<b>52</b>

<b>7</b>	<b>Appendices</b>	<b>54</b>
<b>A</b>	<b>Mutation Operators</b>	<b>54</b>
<b>B</b>	<b>Banking System Code</b>	<b>55</b>
B.1	Account Class . . . . .	55
B.2	CurrentAccount Class . . . . .	57
B.3	SavingsAccount Class . . . . .	58
B.4	LoanAccount Class . . . . .	59
B.5	CurrentWithdrawTest Class . . . . .	60
B.6	SavingsWithdrawTest Class . . . . .	61
B.7	CurrentDoubleWithdrawTest Class . . . . .	62
B.8	CurrentWithdrawLoanDepositTest Class . . . . .	63
B.9	CurrentTransferTest . . . . .	64
B.10	CurrentTransferDepositWithdrawTest Class . . . . .	65
<b>C</b>	<b>Incrementer System Code</b>	<b>66</b>
C.1	Incrementer Class . . . . .	66
C.2	SyncIncRunnable Class . . . . .	70
C.3	LockTest Class . . . . .	70
C.4	InterruptTest Class . . . . .	71
C.5	AwaitSignalTest Class . . . . .	72
C.6	SyncTest Class . . . . .	73

# 1 Introduction

Programmers today have the privilege of extremely powerful computing power and to utilise this power for benefits in performance, they can employ the use of concurrent software. *"Concurrency is the ability of a program to execute several computations simultaneously. This can be achieved by distributing the computations over the available CPU cores of a machine or even over different machines within the same network."* (Mois 2015). Concurrency is a feature that has been present in the Java programming language since nearly its introduction, with a significant update with the release of Java version J2SE 5.0 (Bradbury et al. 2006).

Although concurrency can provide benefits to the performance and execution times of software, with it can come many data issues and unexpected system behaviour, due to the unpredictability of concurrent processes, unless handled properly (Mois 2015). Farchi et al. (2003) discuss the presence of 'bug patterns' that *"describes a commonly occurring error in the implementation of the software design"* relating to concurrent software. In their paper, Farchi et al. (2003) define and categorise a variety of different bug patterns where programmers often make faulty assumptions that their code is protected from errors. By highlighting these common pitfalls, they aim to identify methods for preventing these errors from occurring with effective design patterns.

Following on from this, Bradbury et al. (2006) apply these bug patterns to mutation analysis. Mutation testing is a form of quality assurance for software test coverage, whereby small changes are made to a system's code and the existing test suite is run against this altered code. The tests are evaluated on their ability to detect these changes, or 'mutations'. There currently exist many tools that will automatically insert changes to a code base, run unit tests and provide detailed analysis on the results; Pitest is one such example for Java<sup>1</sup>. However, what is lacking from these tools is support for concurrency related mutations. Bradbury et al. (2006) have created a large range of mutation operators for concurrent features to offer this support, which is especially pertinent considering the many issues surrounding this area and the difficulties of testing unpredictable behaviour. These difficulties will be explored in the following section. The mutation operators are separated into categories based upon the bugs that they are likely to produce, identified by Farchi et al. (2003). The categories target specific concurrency mechanisms and how these can be manipulated or removed to subtly alter the execution of segments of code.

---

<sup>1</sup><https://github.com/hcoles/pitest>

Reviewing the literature reveals that there is a necessity for a concurrent mutation testing tool for Java software and thus lies the purpose of this project. The objectives are to develop a proof-of-concept automatic mutation tool for concurrent software, apply mutations to some sample concurrent systems and evaluate the effectiveness of unit tests at catching these mutations. Four of the mutation operators found in the paper by Bradbury et al. (2006) will be selected and implemented into the mutation tool. Regular expressions will be used to manipulate strings in the source code and provide the mechanism by which the tool applies the mutations. After development is complete, the tool will be tested on two concurrent systems; mutations will be seeded throughout the code and a spectrum of unit tests will be run against the changes. The unit tests will compose of some traditional, non-concurrent JUnit tests and a set of tests that create scenarios that specifically involve concurrent interactions. Prior to any mutations, these unit tests must consistently pass with no errors and their effectiveness will be evaluated on the results produced after mutations have been imposed. If the tests subsequently fail, this indicates that the tests have recognised a change in behaviour of the system and can be considered to be effective. Conversely, if the tests continue to report passes, then they have been unsuccessful at catching the mutations. The results of these test runs will be analysed and reviewed thoroughly.

The structure of this report will be presented as follows. The following section will provide a sufficient overview of concurrency, mutation testing, and a thorough review of the relevant literature relating to concurrent bug patterns and mutation operators by Farchi et al. (2003) and Bradbury et al. (2006), respectively. The methodology and the implemented software will be described in detail in Section 3. Analysis of the test results and the project development process will be evaluated in Section 4. Finally, recommendations for future work and the conclusions will be presented in Sections 5 & 6, respectively. All code referenced throughout this report is displayed in the Appendices.



## 2 Background

The relevant background information required for this project will be presented here, as well as a review of the literature covering concurrent bug patterns and mutation operators. An overview of concurrency in Java and mutation testing for general software will provide the necessary understanding for the motivation of this project.

### 2.1 Concurrency

#### 2.1.1 Threads

Processes are self contained execution environments with private resources; most Java applications only require a single process to run efficiently. Modern computer systems provide multiple cores to process parallel execution of processes. Concurrency utilises this functionality with the use of *Threads*. Threads are light-weight processes that share their resources (memory, open files, etc) with the other threads contained within the process (Mois 2015). With the advancement of technology producing evermore powerful machines, each individual core has the ability to interleave multiple threads as well as the capability of running threads in parallel on separate cores.

Creating threads is lighter on resources than creating new processes and the ability to share resources is beneficial to performance. This is the main motivation for using concurrent code; allowing multiple sections of code to run simultaneously for faster execution times. Another positive aspect of this is the improved responsiveness of a system (Peierls et al. 2005). When one section of a program becomes blocked or slows down, a single threaded system would become unresponsive to the user and would report no information back to explain why. A multi-threaded application would allow a computationally greedy operation to perform in the background without disrupting the rest of the system and remaining responsive to observation and interaction from the user.

Each program has at least one thread, the main thread, created at point of calling the *main()* method, but subsequent threads can be created after this execution. There are two ways to create new instances of threads. The first is to create an object that has implemented the *Runnable* interface and pass this to the *Thread* constructor. The thread will then execute the *run()* method of the

runnable object. The other is to create an object that is a subclass of `Thread` and executing the `run` method pertaining to the object (Oracle 2019a). In both cases, `Thread.start()` must be executed to start a specified instance of a thread. Although, threads are not necessarily run in the order of their `start` execution, the threads are automatically assigned a priority by the Java Virtual Machine (JVM), and scheduled in order of highest priority (Mois 2015). The first instance is a more general and flexible approach since it is implementing the `Runnable` interface, it allows for the class to inherit functionality from another class. This also has the benefit of allowing its own parameters to be passed to the constructor. The second is easier to use in simple applications, but since it is a subclass of `Thread`, there are limitations to what the class can do, as only the parent class `Thread` constructors are available.

If a thread wants another thread to stop then it can invoke an interruption using `Thread.interrupt()`, which will throw an exception message to the interrupted thread. A thread can support its own interruption by invoking the `sleep(long time)` method to stop itself for a specified period of time. After the elapsed time, the thread will resume running, unless an interrupt is called and the thread is terminated. `Sleep` can be used by the user to manage the scheduling of threads if it is prescient for a particular operation and limits the unpredictability of the system behaviour. Similar to a thread sleep, `Thread.join()` can be used for one thread to wait until another has completed its execution. With no specified time, the Thread may wait indefinitely for the other one to terminate.

### 2.1.2 Atomic Operations

An atomic operation is one that is executed and completed all at once or not at all (Oracle 2019a). It is an action that is considered safe from interference from other operations, as it cannot be stopped during execution and the state of the process cannot be changed by another operation. The state is only affected by the atomic operation from the start to finish. A single line of code does not imply atomicity, no matter how deceptively simple it may appear. For instance, the increment operation for some integer, `x++`, is actually three separate operations handled by the JVM. First the value of the integer must be fetched, then it is incremented and finally the new value is written back to memory. This may seem trivial, but if mishandled, this can have unexpected and varying results if it is executed in conjunction with another thread that is accessing the same data. When two threads interleave in such a way, it is referred to as interference. As an example, Thread A and Thread B are tasked to perform the increment operation

Operation	Value of x
Thread A retrieves x	0
Thread A increments x	1
Thread B retrieves x	0
Thread A stores value of x	1
Thread B increments x	1
Thread B stores value of x	1

Table 1: Example of a non-atomic operation,  $x++$ , in two interleaving threads.

on  $x$ . If they were to run consecutively, allowing for complete execution of the first operation before the initiating the next, the value of  $x$  would be expected to have increased by 2. However, if the two operations are executed at the same time, the following scenario could feasibly occur:

1. Thread A retrieves the value of  $x$ ,  $x = 0$ .
2. Thread A increments  $x$ ,  $x = 1$ .
3. Thread B retrieves the value of  $x$  before the Thread A has been given the chance to commit the change of  $x$  to memory and thus,  $x$  remains equal to 0.
4. Thread A stores the result of  $x = 1$ .
5. Thread B increments  $x$  to 1 and stores the result. Despite two instances of incrementing the value of  $x$ , the final value is recorded as 1; the first instance has been overridden by the second.

Table 1 displays this information with the perceived value of  $x$  at each stage.

This is only one small example of interference, but similar occurrences of inconsistent memory issues between threads can crop up throughout concurrent systems unless properly managed. Compounding this is the complexity of other scenarios with a greater number of interleaving threads and manipulation of more data. One method of preventing this is utilising the functionality of synchronized code, which will be explored in the following section.

### 2.1.3 Synchronization and Locks

Synchronization allows for critical sections of code or methods to be executed atomically in relation to code in other threads. The power of scheduling operations and resource management is passed to the user (Silberschatz et al. 2013). The `synchronized` keyword blocks other threads from executing simultaneously during its execution to ensure that the state of the system is only affected by the actions within the synchronized block. Synchronization can be applied to a specific set of actions in a block, specifying the resource to be restricted or to an entire method. The two mechanisms of using the `synchronized` keyword are shown in Figure 1.

**Synchronized Block:**

```
public class SomeClass {
    private Integer x = 0;

    public void someMethod() {
        synchronized (x) {
            //do something
        }
    }
}
```

**Synchronized Method:**

```
public class SomeOtherClass {
    private Integer x = 0;

    public synchronized void synchronizedMethod() {
        //do something
    }
}
```

Figure 1: Synchronized block and synchronized method

Restricting the whole method will have a more dramatic effect on the performance as no other threads can run concurrently. Applying synchronization to the increment example would solve the observed problem of memory consistency and would guarantee the expected result to occur.

The *synchronized* keyword achieves the desired affect by imposing a lock on the synchronized block of code that prevents access to the contained object's fields. After the synchronized block has terminated, the lock is released and normal thread

```

public Lock lock = new ReentrantLock();
public int x = 0;

public void method() {
    lock.lock();
    x++;
    lock.unlock();
    System.out.println("Count = " + x);
}

```

Figure 2: Example of a Reentrant lock in use with the *lock* and *unlock* methods.

scheduling will resume. Reentrant locks can be used to similar effect to the synchronized keyword, allowing a thread to obtain and release a lock during execution of critical code, much like a synchronized block of code. Initially, a *lock* object is created and then the critical section of code can be protected by calling the *lock()* method immediately before and the *unlock()* method immediately after. A demonstration of this is shown in Figure 2.

#### 2.1.4 Liveness

The liveness of an application expresses its ability to execute without complications in an efficient manner. There are some issues that can interfere with the liveness of an application by slowing it down or even freeze functionality altogether. A deadlock can occur when two threads have locked separate resource, but are waiting on the other thread to release their lock to continue execution.

Thread 1: locks resource A, waits for resource B  
 Thread 2: locks resource B, waits for resource A

In this instance, both resources are locked and unavailable for access until the other is released; the program will perpetually hang in an inescapable catch-22.

#### 2.1.5 Executor Service

Previously, concurrency examples provided have only involved two threads that are relatively simple to follow. In many cases, a program may wish to utilise many more threads for a significant boost in performance. The Java concurrent package

offers the Executor Service to help maintain many instances of threads, a thread pool, by abstracting the management and construction of threads from the main program. This can be an invaluable tool for ensuring computation resources are not wasted by the constant creation of new threads for each concurrent action. Instead, a pool of threads is created at once and the threads will wait until they are required. After performing the necessary action, the thread will go back into a state of waiting for a request for work. The size of the pool has a specified limit, to prevent a runaway of thread creation; when all the threads in a pool are currently in use, the executor server will wait to assign work to the next available thread. Creating a thread takes up time and resources, it is far more efficient to recycle previously used threads.

## 2.2 Mutation Testing

Mutation testing is the process of seeding errors throughout a system's codebase, to observe the effecting behaviour and evaluate the effectiveness of the present testing coverage (Adrion et al. 1981). A *mutation operator* is a generalised rule, which describes the changes that will be made to a specific segment of code. The result of applying an operator to a code segment is known as a *mutant* (Ammann and Offutt 2017). The premise of mutation testing is simple: if the previously implemented software tests have been designed adequately, then they should recognise the changes to the system behaviour and certain tests should fail accordingly. The mutation is referred to as killed in this instance. This is the ideal outcome when a mutation is applied to a system, as it suggests that the tests have been designed effectively and the coverage is sufficient. However, if a mutation survives by circumventing the tests, this indicates that the tests are either flawed or the test coverage has gaps and not all of the behaviour has been accounted for. This is a useful tool for highlighting weaknesses in test coverage; a full suite of unit tests that successfully pass when run on a system only informs the user that those specific function behaviours are expected. Although it provides no information on the behaviours that have been missed. Mutation testing aides identifying these areas of code that have been overlooked by applying many different mutations throughout the whole system, which should disrupt as much expected behaviour as possible. Thus, when a mutation is not successfully killed by any unit test, the location of the altered code will indicate that there is improvement to be made in the test coverage related to the affected area.

One example of a renowned mutation testing tool is Pitest (PIT), which combines traditional line coverage with mutation coverage, to offer a comprehensive

and fast testing environment for Java applications. The list of mutation operators it provides is extensive, covering relational operators (e.g. `<`, `<=`, `>`, `>=`), mathematical operators (e.g. `+`, `-`, `*`, etc.), logic statements a variety of different common method calls and return values (Coles 2019). A mutation usually consists of altering a small section of code or removing a section entirely. For example, a conditional boundary operator would make the following mutation in Figure 3.

<b>Original Code:</b>	<b>Mutant:</b>
<pre>if (a &lt; b) {     //do something }</pre>	<pre>if (a &lt;= b) {     //do something }</pre>

Figure 3: Relational operator mutation

The code now has a slightly different meaning. Designing effective unit tests involves specifically testing code at such boundary cases. The behaviour of the code would be monitored for values of  $a$  when less than  $b$ , equal to  $b$  and greater than  $b$ , with a separate test for each scenario. Prior to the mutation, the if statement would not be executed during runtime for when  $a$  is equal to  $b$ , but after the mutation the if statement would be entered. Thus, a unit test that previously would pass for this scenario should fail due to the alteration and the mutation could be considered successfully killed. PIT will apply many of these mutations on the byte code generated after compilation, instead of on the source files. This produces significantly faster runtimes. After mutation, PIT will automatically run the new java files against the designated unit tests and produce a set of results detailing the fates of each mutation. The varying states explained below:

**Killed** - The mutation was successfully discovered by the presence of a failed unit test.

**Lived** - The mutation was unsuccessfully discovered with no failed unit tests.

**No coverage** - The mutation lived because of a lack of unit tests covering the relevant mutated section of code.

**Non-viable** - The mutation affected the Java bytecode such that the JVM could not load the file.

**Timed out** - The mutation created an infinite loop so execution of the file could not terminate.

**Memory error** - The mutation increased “the amount of memory used by the system or the result of the additional memory overhead required to repeatedly run your tests”.

**Run error** - The mutation caused the file to be unable to run, similar to non-viable mutations. (Coles 2019)

The results detail which mutations were applied, identify which tests managed to kill mutations and produce the ratio of successfully caught mutations to the total number of seeded errors. This ratio is known as a *mutation test score* (Bradbury et al. 2006). Although PIT offers a wide range of mutation operators, it is lacking in support for concurrent systems. Performing unit tests on multi-threaded code is not as straightforward as single-threaded.

## 2.3 Regular Expressions

The theory of mutation testing has been presented, however the mechanism by which mutations are applied is not a simple process with a single approach. The method used in this project is a high-level approach, utilising regular expressions to identify and manipulate strings. This is a different approach to the method seen in the Pitest mutation tool; a more sophisticated and technically complex method operating on bytecode. For the size and scope of this project, performance is less of a concern, so manipulating source code is a viable option.

Regular expressions, regex for short, are a syntactic description of a pattern, often used to search or manipulate strings in a text (Oracle 2017). Exact strings can be found with ease, but the true power behind regular expressions is the ability to search for generic patterns and manipulate any matches returned. The *java.util.regex* package allows a user defined pattern to be interpreted and will find any matches within a given text. This is primarily achieved using the *Pattern* and *Matcher* classes. Inputting a regular expression into a *Pattern* as a parameter will create a compiled version of the regex. The syntax of a pattern is built up of special character constructs that can match with independent characters or a defined range of characters. For example, a regex pattern could be used to search for a date in the following format DD month YYYY, i.e. 20 July 1969. This format has strict rules specifying that a date must consist of two digits followed by a word and finally four more digits. The regex can be made more complex by imposing more rules limiting the range of numbers for the day section to be between 1-31; the month section to only consist of the exact strings for the calendar months and



the year section to be greater than 0000. All of this is achievable with the Java regex package. However, to keep it simple, the following regex example will only look for the basic two digits-word-two digits:

$$(\backslash d\{2\})(\backslash s)([a-zA-Z]+)(\backslash s)(\backslash d\{4\}) \quad (1)$$

$\backslash d\{2\}$ ,  $\backslash d\{4\}$  - exactly 2 or 4 digits, respectively

$\backslash s$  - a single whitespace character

$[a-zA-Z]^+$  - one or more letters in the range of a to z, lower or upper case

The brackets separates the regex into groups that can be manipulated in isolation from the rest of the matched expression. A `Matcher` object can then be created to compare a character sequence against the pattern and return any matches. The `find()` method attempts to find the next matching sequence in the input. The `group(int group)` method returns the sequence that was matched by the specified group, identified in order of appearance in the regex, e.g. `([a-zA-Z]^+)` is group 3. Finally, if the user wishes to replace any part of a matched string, the `replaceFirst(String replacement)` and `replaceAll(String replacement)` methods will replace either the first matched substring or all matching substrings, respectively, with a specified replacement string. A full API for these classes is provided by Oracle (2019b).

## 2.4 Concurrent Bug Patterns

During the software development life cycle, it is vitally important to contribute a significant portion of effort into the architectural design of the software. A well designed system will provide a solid foundation in avoiding unforeseen faults throughout development. Farchi et al. (2003) present a systematic approach to preventing certain concurrency related errors in their research on Concurrent Bug Patterns (Farchi et al. 2003). *“Design patterns are solutions to recurring problems in a given context. A design pattern accentuates the positive, i.e., how to solve a recurring problem well.”*

This is a general concept, originally used to describe physical construction, but is equally applicable to software development (Gamma et al. 2015). However, poor design patterns can have the reverse effect and introduce their own set of errors. This gives rise to what is known as a bug pattern: *“A bug pattern is an*

*abstraction of a recurring bug. In other words, a bug pattern is a literary form that describes a commonly occurring error in the implementation of the software design.”*

By identifying common concurrency errors made by developers, Farchi et al. (2003) have categorised 8 different bug patterns. In their systematic approach, they offer a more technical definition of a bug pattern in a program,  $P$ , relating to potential number of interleavings between threads in a concurrent system,  $I(P)$ , and the maximum number of interleavings the system can have whilst remaining correct,  $C(P)$ . A concurrent bug pattern can be found within the range  $I(P) - C(P)$  Farchi et al. (2003). Typically, bugs will occur due to a faulty assumption by the developer, separated into the following three categories:

1. *”A code segment is mistakenly assumed to be undisturbed, implicitly or explicitly, by other threads;*
2. *As a result of the mistaken assumption that a certain execution order of concurrent events is impossible;*
3. *When a code segment is mistakenly assumed to be nonblocking.”*, (Farchi et al. 2003).

Farchi et al. (2003) provide many examples in each category, but only the relevant bug patterns will be explored in the following sections, separated into the categories defined above. Due to the limited scope of the project, instances of many concurrent keywords are not present, meaning that some bug patterns are not available for exploration. All the bug patterns that have the potential to be found are presented here.

### **2.4.1 Unprotected Code**

Concurrent code can be considered to be protected when only a single thread is executing a concurrent event between the first and last events in the code segment. When multiple threads begin executing concurrent code simultaneously, errors are bound to arise.

#### **Nonatomic Operations Assumed to be Atomic Bug Pattern**

This bug relates to the example in Section 2.1.2, wherein a developer falsely assumes that a fragment of code is an atomic operation and therefore protected. On

a surface level, a code fragment may appear to be executed as a single operation, but the bytecode translation consists of more operations.

### **Wrong Lock or No Lock Bug Pattern**

This pattern can occur when one thread has locked an action but other threads attempt to acquire a different lock for a concurrent action. The other threads will either successfully obtain the wrong lock or don't obtain any lock. Thus, the code is unprotected and susceptible to interference from interleaving threads.

## **2.4.2 Unexpected Interleavings**

In these scenarios, the programmer has assumed an interleaving between threads to be impossible, often due to considering the computation time of a certain action to be fast enough that it will not overlap with another concurrent action. Generally, this is considered bad practice as it is often difficult to predict the length of time for a process to complete, which can also vary between executions.

### **Sleep() Bug Pattern**

A programmer might understandably attempt to control the scheduling of thread execution by introducing delays, utilising the *sleep()* method, and specifying a time they have deemed to be sufficient for complete execution of certain critical sections. Instead, the *join()* method would be more appropriate in this circumstance.

Farchi et al. (2003) cover another example of an unexpected interleaving bug pattern involving the *notify()* and *wait()* methods. However, the concurrent systems tested in this project do not contain any instances of these methods and thus, the *notify()* bug pattern will not be covered in this review.

## **2.4.3 Blocking Code**

In some circumstances, a segment of code can have an unexpected behaviour in a thread that blocks other threads from executing, resulting in the system hanging indeterminately. This obviously can have a very drastic effect on the performance of a program.

### **Blocking Critical Section Bug Pattern**

After execution of a critical section in a thread is complete, it is expected to relinquish control and allow other threads to execute. If the correct procedure for this has been overlooked then other threads are left waiting for the first thread to terminate; an event which may never happen.

Again, only one of the patterns in this category can be found in the concurrent systems and is covered in this section.

## 2.5 Concurrent Mutation Operators

Bradbury et al. (2006) have comprehensively designed a set of mutation operators specific to the concurrent functionality and the bug patterns identified by Farchi et al. (2003). The focus of their research is in response to the updated concurrent functionality introduced in the J2SE 5.0 version of Java. Synchronization can now be implemented using explicit locks, semaphores, barriers, latches and exchangers. Support for these various concurrent operations persists through the more recent versions of Java with some minor updates and revisions. In total, Bradbury et al. (2006) produce 22 different mutation operators, each of which are associated with a number of different concurrent methods. The operators are split into groups relating to the concurrent bug patterns described in the work of Farchi et al. (2003).

Mutation analysis for non-concurrent systems has been covered extensively in research and made available through a variety of different tools such as PIT, Jumble<sup>2</sup> and Jester<sup>3</sup>. With their mutation operators, Bradbury et al. (2006) aim to help improve the quality and development of concurrent Java applications by making programmers aware of the various pitfalls surrounding concurrency. The operators have been divided into five categories:

1. *"Modify parameters of concurrent methods"*
2. *Modify the occurrence of concurrency method calls (removing, replacing and exchanging)*
3. *Modify keywords (addition and removal)*
4. *Switch concurrent objects*
5. *Modify critical regions (shift, expand, shrink and split)" (Bradbury et al. 2006)*

---

<sup>2</sup><http://jumble.sourceforge.net/>

<sup>3</sup><http://jester.sourceforge.net/>

Some of these operators are modified versions of existing operators, whereas some are novel. 1-3 of the above categories will be explored in the following sections. Categories 4 & 5 are not covered since none of the mutation operators from these categories are implemented in the tool developed for this project. The specific operators that have been implemented will be fully explored in the Methodology section of this report. The full list of mutation operators can be found in Table 14 in Appendix A (Bradbury et al. 2006).

### **2.5.1 Modify Parameters of Concurrent Method**

Altering the parameters of a method in any way can cause a dramatic difference to the original intention when calling the method. The operators in this category aim to do just that for any concurrent method or methods related to threads. This can involve changing the value of an input parameter or removing a parameter altogether, assuming the method has an overloaded version to support this removal.

### **2.5.2 Modify the Occurrence of Concurrency Method Calls**

In contrast to the previous category of modifying method parameters, this set operates on the method calls themselves. The mutation can manifest in three forms: a method call can be removed, replaced or exchanged with a similar method.

### **2.5.3 Modify Keywords**

This category is similar to the previous type, but focuses solely on the addition or removal of certain concurrent keywords, such as *static*, *synchronized*, *volatile* and *finally*. These keywords affect the behaviour of classes, methods and variables, thus modifying them may have significant effects when calling a mutant version.

## 3 Methodology

The background information has been presented and the literature surrounding Mutation testing operators has been explored. The motivation for the project should now be clear. Although there are mutation testing tools available for Java applications, they lack support for concurrency. Thus, the aim of the project is to build a mutation testing tool that will automatically apply mutations specific to concurrent operations, utilising the operators provided by Bradbury et al. (2006). This will provide support for programmers to analyse the effectiveness of their unit tests for multi-threaded functionality.

This Methodology section will describe the development process in detail and the steps taken to ensure the mutation testing tool is robust.

### 3.1 Selected Mutation Operators

Before development of the software for the mutation testing tool, a small selection of mutation operators from Bradbury et al. (2006) were chosen to be implemented into the tool. This section describes the selected operators: *Modify Method-X Timeout*, *Modify Synchronized Block Parameter*, *Remove Thread Method-X Call*, and *Remove Concurrency Mechanism Method-X Call*. These four operators target commonly used concurrency features and the mechanisms by which they apply their respective mutations are straightforward. Thus, they proved to be the ideal candidates for implementation by maximising the number of potential method mutations and required the least amount of time to develop. Since the latter two operators both involve removing method calls, only one programming function was necessary to apply these mutations to the list of different methods.

#### 3.1.1 MXT - Modify Method-X Timeout

The MXT operator falls under the category of *Modify Parameters of Concurrent Method*, found in Section 2.5.1. The objective of this operator is to modify the time parameter in the methods *wait(long time)*, *await(long time)*, *sleep(long time)* and *join(long time)* (Bradbury et al. 2006). The *wait*, *await* and *join* methods all have an overloaded equivalent without the time parameter, meaning that the mutation can remove or modify for varying effects. Removing the time parameter from the *wait* method forces the current thread to remain inactive until a *notify()*

or *notifyAll()* method is called to release the interruption. The *await* method is similar to the *wait* method, but is instead released by a *signal()* or *signalAll()* method call in another thread. The *join* method is called on a thread and will wait until it has completed execution or until the specified time has elapsed, after which, the code following the *join* call will be executed in the same thread. The *sleep* method will interrupt the current thread for the specified time. The mutation that has been implemented in the tool is to either remove the time parameter, if applicable, or to reduce the time by half. An example of this type of mutation can be seen in Figure 4.

<b>Original Code:</b> <pre>long time = 100; try {     wait(time); } catch ...</pre>	<b>MXT Mutant:</b> <pre>long time = 100; try {     wait(time/2); } catch ...</pre>
--	---

Figure 4: MXT mutation (Bradbury et al. 2006)

In this example, prior to the mutation, the wait time had been set to a sufficient length to allow other threads to fully execute any concurrent operations that may interfere with mutually accessed resources. However, cutting this time in half may not leave enough time for another thread to complete execution. Various similar problems can occur when applying the MXT mutation to the other concurrent methods mentioned above. These errors will be discussed in the Results section, for each method tested.

### 3.1.2 MSP - Modify Synchronized Block Parameter

The MSP is another *Modify Parameters of Concurrent Method* operator that alters the parameter of a synchronized block. The *synchronized* keyword is applied to an object that the programmer wishes to be thread safe when executed by blocking other concurrent actions from taking place. The mutation made by this operator aims to replace the synchronized object parameter with another object (Bradbury et al. 2006). The simplest method of achieving this is to replace the object with the keyword *this*. An example of this is given in Figure 5.

The keyword *this* refers to the current object: an instance of the class in which the method has been defined. Thus, the effects of changing the lock object to *this*, the lock object becomes no longer thread safe and executing the critical section of code leaves the lock object susceptible to unpredictable behaviour when

**Original Code:**

```
private Object lock = new Object();

public void syncMethod() {
    synchronized(lock) {
        \\Do something
    }
}
```

**MSP Mutant:**

```
private Object lock = new Object();

public void syncMethod() {
    synchronized(this) {
        \\Do something
    }
}
```

Figure 5: MSP mutation (Bradbury et al. 2006)

interleaving with other threads.

**3.1.3 RTXC - Remove Thread Method-X Call**

The RTXC belongs to the *Modify the Occurrence of Concurrency Method Calls* category in section 2.5.2. This mutant simply removes calls to the Thread methods *wait()*, *join()*, *sleep()*, *yield()*, *notify()*, and *notifyAll()* (Bradbury et al. 2006). Figure 6 provides an example for this mutation.

**Original Code:**

```
try {
    wait();
} catch ...
```

**RTXC Mutant:**

```
try {
    //wait() removed
} catch ...
```

Figure 6: RTXC mutation (Bradbury et al. 2006)

Removing a call to any of the Thread methods also removes control from the user in their ability to schedule thread execution. The behaviour of the system is likely to change from the original expectations.



### 3.1.4 RCXC - Remove Concurrency Mechanism Method-X Call

The final mutation operator that was selected aims to remove a variety of different concurrent methods: *lock()*, *unlock()*, *signal()*, *signalAll()*, *acquire()*, *release()*, *countDown()*, and *submit()* (Bradbury et al. 2006). It utilises the same removal mechanism as the previous operator, RTX, and is also in the same category, *Modify the Occurrence of Concurrency Method Calls*. Potentially the most significant method removals are for the *lock* and *unlock* methods. This is shown in Figure 7.

**Original Code:**

```
private Lock lock = new ReentrantLock();

lock.lock();
try {
    ...
} finally {
    lock.unlock();
}
```

**RCXC Mutant:**

```
private Lock lock = new ReentrantLock();

lock.lock();
try {
    ...
} finally {
    //lock.unlock() removed
}
```

Figure 7: RCXC mutation (Bradbury et al. 2006)

Altering a lock on an object will have a similar behaviour to mutating a synchronized block; the object becomes no longer thread safe. Removing an unlock will prevent this thread from releasing the lock and will prevent any other threads from acquiring the lock to begin execution. Referring to work of Farchi et al. (2003), this can result in a *Blocking Critical Section Bug* described in section 2.4.3.

## 3.2 Mutation Tool

With the four mutation operators identified above, the core functionality of the mutation testing tool could be designed. The main objective of the software is as follows:

- Select an input file, a mutation operator and a method to be mutated
- Count the number of instances of the selected method in the input file
- Randomly select a match and perform the mutation
- Write the mutation to a new file

The system consists of two classes: the Mutator class contains the methods that perform the above functionality, and the MutatorRunnable class creates a Mutator object, passes the necessary input and executes the mutation. The Mutator class accepts a File object as its only parameter. The mutation is performed by calling the *replaceMutation()* method.

When running the mutation tool, the user is prompted to select an input file, an operator and a method they wish to be mutated. The tool will recognise if this combination is suitable by verifying that the input are compatible. If the user has attempted to input an incompatible pair of method and operator, then an error will flag up indicating this. This is achieved by storing a string of each operator and method in an ArrayList related to each operator. When the user inputs their desired mutation operator and method, the strings will be compared against the ArrayList contents to verify that their options are suitable. If the user selects a successful pair then the tool will begin to search the input file for any matches against the regex pattern. The matches will be returned to the user with their respective character location in the input file. The mutation is designed to only be applied to one match at random, if there are multiple matching methods.

An algorithm was created to achieve this random selection by applying the following method. A count of the matches is stored as a local variable. The first match is encountered and a *Random()* function chooses a number between 1 and the count. If the random number matches the count then the first match will be selected to be mutated. The chances of this happening are 1/count; the algorithm provides an equal chance of mutation to each match. When the random number does not match, the count is reduced and the process is repeated for each subsequent match.

The following regex pattern was designed to match with any combination of mutation operator and associated method.

$$(\backslash s^* \cdot^* \backslash b \text{ method } \backslash () (\backslash w^*) (\cdot^*)) \quad (2)$$

$\backslash s^*$  - zero or more whitespace characters e.g. the beginning of a line  
 $\cdot^*$  - zero or more of any character e.g. any number of chained methods  
 $\backslash b$  - a word boundary e.g. a full stop indicating the calling of the selected method or the beginning of the line  
 $\backslash ($  - a bracket e.g. the beginning of the method parameter  
 $\backslash w^*$  - zero or more of any word characters e.g. any number of parameters  
 $\cdot^*$  - zero or more of any character e.g.

The regular expression was generalised so that only one pattern was required, instead of a separate pattern for each combination of method and mutation operator. The pattern is separated into the three groups in red brackets. In general, the first group identifies the method call, the second identifies the parameters and the third identifies any syntax following the end of the parameters e.g. `)`, `{`, `,`, `;`, etc. Before the pattern is compiled, the regex is stored as a string with the input method stored as a string variable. Thus, for any method that the user wishes to mutate, the regex string can be easily updated and then compiled as a Pattern object. Group 1 allows for the tool to easily find the specified method by the user. Group 2 is the main target for two of the mutation operators as they aim to alter the parameters. The MXT operator accesses group 2 and appends the parameter with `"/2"` to divide the specified length of time by 2 or simply remove the parameter altogether. The MSP operator accesses group 2 and replaces the parameter object with the keyword *this*. The operators RCXC and RTXC both remove the entire method call by deleting the entire matched pattern from the code.

Once a match has been selected, the mutation will be applied to the relevant section of code using the `String.replaceFirst(regex, replacement)` method. The limitation of this method is that it will perform a replacement only on the first instance of a match within a string. To overcome this, the full input file is split into two separate strings for each match: one before, excluding the match and one after, including it. This ensures that the current selected match is the first occurrence in the second string. The mutation can therefore be applied to any of the method matches in a file. After the mutation, the two strings are stitched back together to recreate the original file with only the desired alteration. The mutant is written to a new file within the project. Note that the original file remains completely intact and the mutant is a separate file.

For example, Figure 8 shows a code segment that has two wait methods that would match against a mutation operator such as the MXT. In this scenario, the mutation tool has identified that there are two matches. On the first iteration, when deciding which match to apply the change to, it has split the code into two strings up to the first match and after. If it decided to mutate the second match, then the two strings are shifted to before and after the second instance of wait.

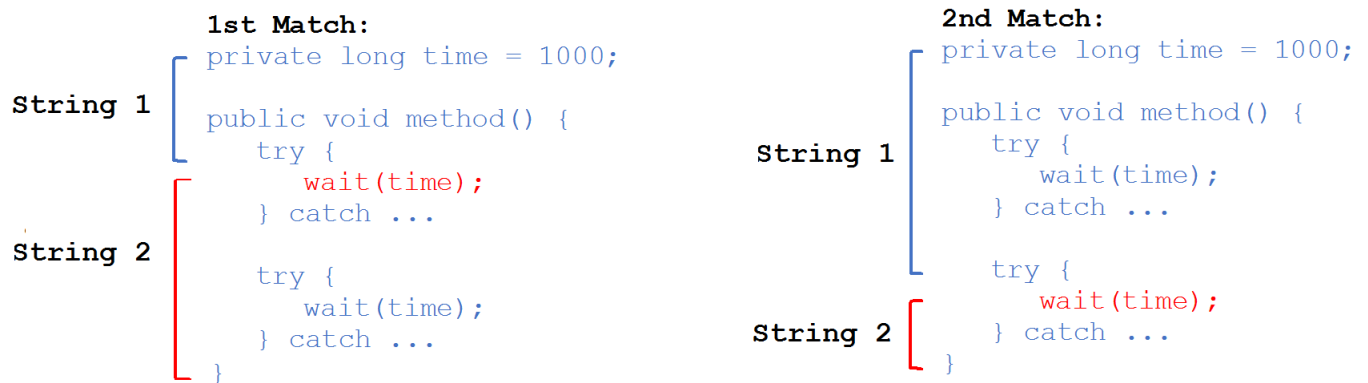


Figure 8: Input code is split into two strings, before and after a method match.

### 3.3 Concurrent Software

With the mutation testing tool built and the mutation operators fully implemented, to ensure the tool worked effectively, it needed some concurrent systems to test on. A typical banking system was selected after having identified that it contained a sufficient number of different concurrent features that were compatible with the four mutation operators the tool had to offer. The second, the Incrementer system, was developed to match the concurrent features found in the banking system but with different core functionality. Applying the tool to a variety of software systems provides greater verification and assurance of the tool's effectiveness. The two systems and the unit tests designed for each will be explored in detail in the following sections.

### 3.3.1 Banking System

The *Banking* system contains four different *Account* classes that provide different functionality for slightly different purposes when handling money<sup>4</sup>. The *Account* class is an abstract parent class that the *CurrentAccount*, *LoanAccount* and *SavingsAccount* all inherit from. The main difference between the account classes is their specific method of withdrawing or depositing money. Each method also has an associated *Runnable* class to allow for multi-threaded execution. The *Account* class holds generic methods for depositing and transferring money, as well as the base constructor which defines the initial balance and name of the *Account* object. The following methods all contain concurrent functionality in an attempt to prevent interference from interleaving threads: *deposit(double amount)*, *withdraw(double amount)* and *transferMoney(double amount, Account recipient)*. These methods and their functionality will be explored below. The code for all *Banking* system classes can be found in Appendix B.

#### **Account - deposit(double amount)**

This method aims to deposit the specified amount of money into the currently accessed *Account* object. A lock on the critical section of code is introduced to prevent other threads from performing any interfering actions. After updating the balance, a *signalAll()* is called on the concurrent condition, *fundsAvailableCondition*, which informs other threads that the *Account* object now has a positive balance. Finally, the lock is released to allow other threads to execute concurrently. The code can be found in Appendix B.1.

#### **LoanAccount - deposit(double amount)**

The *LoanAccount* overrides the parent deposit method to support the different style of account. The *LoanAccount*'s purpose is to allow the user to take a loan of a specified amount of money and deposit that money into a *CurrentAccount*. The initial balance of the *LoanAccount* is then the negative of this amount. Money can be deposited into the *LoanAccount* to pay off the loan and if the balance becomes greater than £0, the positive balance will be deposited into the *CurrentAccount* from before. The deposit method utilises concurrency through the lock system and calls the parent version of the deposit method detailed above. The code can be found in Appendix B.4.

---

<sup>4</sup>The four Banking system Account classes were taken from a Strathclyde undergraduate Computer Science class on concurrency.

### **CurrentAccount - withdraw(double amount)**

The *withdraw* method has the simple objective of removing money from an *Account*, however, there are some rules imposed on when this is allowed. The *CurrentAccount* constructor sets an overdraft limit of £50, meaning that the account cannot have less than -£50 available after any transaction. If the user attempts to withdraw an amount of money that would leave the account with less than this limit then the thread will wait until there are sufficient funds in the account by calling an *await(time)* on the *fundsAvailableCondition*. The thread will resume when a *signalAll()* method is called in another thread, such as the *deposit* method, or until the elapsed time has passed. In the instance of time running out and there still not being enough money in the account, the thread will be interrupted using *Thread.interrupt()* and the withdrawal will not take place. After either circumstance, the method will release control of the lock it has. The code can be found in Appendix B.2.

### **SavingsAccount - withdraw(double amount)**

The *SavingsAccount* version of *withdraw* has the same functionality as the *CurrentAccount* version with the addition of imposing a fee on each withdrawal. The user selects an amount to be withdrawn from the *SavingsAccount* but this amount plus a predefined fee will be withdrawn. The concurrent features are the same as those found in the *CurrentAccount* method. The code can be found in Appendix B.3.

### **Account - transferMoney(double amount, Account recipient)**

The *transferMoney* method withdraws money from one *Account* and deposits that amount in another *Account*, using the respective methods. Thus, it does not have any unique concurrent issues of its own, although it will encounter the same problems that can be found in the *deposit* and *withdraw* methods. A successful transfer will only take place if the withdrawal is successful and returns true. The code can be found in Appendix B.1.

Tables 2 & 3 highlight every concurrent method that occurs in each of the Bank System methods described above. This information was then used to design the effective unit tests for all concurrent behaviour within the Bank System, which will be explored in Section 3.4.

		Concurrent Feature					
		lock	unlock	signalAll	interrupt	await	newCondition
<b>Bank System Method</b>	Account constructor						
	Account deposit						
	Account transferMoney						
	LoanAccount constructor						
	LoanAccount deposit						
	CurrentAccount withdraw						
	SavingsAccount withdraw						

Table 2: Concurrent features that occur in the Banking System methods. Green boxes indicate that the concurrent method is directly called. Blue boxes indicate that the concurrent method is indirectly called by calling another method. For example, the Account transferMoney method calls a deposit and a withdraw method which contain the concurrent methods signalAll, interrupt and await.

		Concurrent Feature					
		lock	unlock	signalAll	interrupt	await	newCondition
<b>Bank System Class</b>	Account						
	CurrentAccount						
	SavingsAccount						
	LoanAccount						

Table 3: Concurrent features that occur in the Banking System classes. Green boxes indicate that the concurrent method is directly called. Blue boxes indicate that the concurrent method is indirectly called by calling another method.

### 3.3.2 Incrementer System

The Incrementer system was designed with the specific purpose of containing all of the same concurrent features found in the Banking System. A list of these methods can be found in Table 4. The core functionality of this system is basic and mostly involves exploiting the non-atomic action of incrementing an integer,  $x$ , using the operation  $x++$ . Section 2.1.2 explains this operation in detail. This system consists of one main class, `Incrementer`, with a variety of similar methods that perform the  $x++$  operation but utilising different concurrent features. The motivation behind creating this system was as a proof-of-concept and to help reproduce the results of performing unit tests on the Banking System. Thus, the Incrementer system is not very complex or pertains to any practical usefulness in its functionality. Each of the Incrementer concurrent methods will be explored below. The code for the *Incrementer* class can be found in Appendix C.1.

#### **Incrementer - increment(int inc)**

The base *increment* method allows the user to enter an amount to increase a global integer count by. A delay is introduced between each increment using the *Thread.sleep* method. There are no safety mechanisms implemented, such as a lock or synchronized block and the time delay was set to 750 ms to artificially provide enough time for interference to regularly occur from concurrently executing threads. This is the only concurrent feature of this method.

#### **Incrementer - incrementLocked(int inc)**

The *incrementLocked* method is the thread safe version of the base increment by introducing a *lock* and *unlock* call before and after an *increment* call.

#### **Incrementer - interruptInterrupt(int inc)**

This incrementing method invokes an interruption on the current thread with a *Thread.interrupt()* call if the count mod 5 is equal to 0. The count will only be incremented if a *Thread.interrupted()* call returns true. With unmutated code, this will always be true but applying the RTXC mutation operator on the interrupt call will cause the method to behave incorrectly.

#### **Incrementer - incrementAwait(int inc)**

The *incrementAwait* method operates on a similar premise to the *fundsAvailableCondition* seen in the Banking System withdraw methods. This method will



execute an increment if an *await* call is interrupted by a *signalAll*, or if the count is greater than zero at the start of execution. Otherwise, no increment will occur.

### Incrementer - incrementSignal(int inc)

This method is similar to the *incrementLocked*, however, before releasing the lock, it will call a *signalAll* on the global condition to alert the *incrementAwait* method that it can continue execution.

### SyncIncRunnable - run()

In addition to the concurrent methods found in the Banking system, the synchronized keyword was included into the Incrementer system as it is such a key feature of concurrency and the *MSP* mutation operator was implemented into the mutation tool. The *SyncIncRunnable* class is separate from the *Incrementer* class and is simply the mechanism by which threads can execute multiple instances of a method on the same object. The *synchronized* keyword is applied to an *Incrementer* object, which can be replaced with the *this* keyword by the *MSP* mutation. In the synchronized block, the *Incrementer* object calls the basic *increment* method. This will have a similar effect to calling the *incrementLocked* method. The code can be found in Appendix C.2.

		Concurrent Feature							
		lock	unlock	interrupt	signalAll	await	newCondition	sleep	synchronized
Incrementer System Method	Incrementer Constructor								
	increment								
	increment- Locked								
	increment- Interrupt								
	increment- Await								
	multiplier- Signal								
	SyncInc- Runnable								

Table 4: Concurrent features that occur in the Incrementer System methods. Green boxes indicate that the concurrent method is directly called. Blue boxes indicate that the concurrent method is indirectly called by calling another method.

### 3.4 Unit Tests

Having identified a variety of concurrent methods and implemented operators to mutate these instances throughout a system, the next step in this project was to design effective unit tests to sufficiently evaluate all the functionality of the implemented systems. Typically, for Java systems, the JUnit testing framework is sufficient for creating unit tests. However, JUnit lacks support for executing multiple threads concurrently and thus is not suitable for evaluating most of the software developed in this project. As a result of this, another approach to unit testing had to be taken; *assert* statements provide similar functionality to that found in the JUnit framework without the same limitations relating to concurrency.

The key to writing effective unit tests is to identify each individual component of a code segment and isolate its behaviour under all possible outcomes. This involves creating tests that guarantee the execution of each different path within a method, such as the branches in an *if/else* block. As well as this, it is important to observe the behaviour of a system in boundary cases where errors are most likely to arise. An example of this would be testing the statement *if( $x < y$ )* for each of the cases where  $x$  is less than  $y$ ,  $x$  is equal to  $y$  and  $x$  is greater than  $y$ . Applying these principles to all sections of code will add confidence of a strong test coverage.

Despite the limitations with concurrency discussed above, JUnit provides an extremely effective environment for testing single threaded code and ensuring that each individual component behaves as expected. A JUnit test class was created for each of the Banking system classes and the Incrementer class. All methods were tested, multiple times if they produced different outputs from varying circumstances. Even the concurrent methods were tested to ensure they behaved in the expected manner when they were executed without interaction from other threads.

The unit tests written for concurrent actions were of a slightly different structure and purpose to the traditional unit tests using JUnit. The concurrent tests focus more on evaluating the specific concurrent features of the systems. Several scenarios were identified that could potentially lead to interfering threads attempting to access the same resource. In the Banking system, most of the methods were attempting to access and modify the balance of an Account object. If two threads attempted to deposit or withdraw money whilst another thread was modifying the balance, then the final balance query may be different from the expected value, unless the methods have the necessary precautions in place. Section 2.4 details the various issues that can occur when a system is not properly safeguarded from interfering threads. A total of seven concurrent unit tests were created for the

Bank System and four for the Incrementer system. The two systems in their original states without mutations needed to be verified as thread-safe by passing all of their respective concurrent unit tests. Each test produces at least one scenario for interleaving methods, many of them produce more as the execution order of the threads can vary. These scenarios will be described below.

### 3.4.1 Banking System Concurrent Tests

#### CurrentWithdrawTest

This test performs a withdrawal in one thread and a deposit in another, from a *CurrentAccount*. The aim of the first scenario is for the *withdraw* to execute first and to encounter the situation where the amount of money attempting to be withdrawn is over the limit and there is not enough money to perform the withdrawal immediately. The *withdraw* thread will then wait until the account has enough money to make a successful withdrawal. Whilst this thread is interrupted, the other thread will be allowed to execute and will make a deposit to the shared *CurrentAccount*. After the second thread executes a call to the *signalAll* method, the first thread will resume execution and the account will have enough money available to complete the withdrawal. The other scenario involves the deposit transaction executing first and the withdraw transaction after. The *withdraw* method no longer has to wait for funds to become available. Without any mutations, this test should pass every time as sufficient steps have been taken to prevent any interference from the two threads. The code can be found in Appendix B.5.

#### SavingsWithdrawTest

The *SavingsAccount* has a near identical *withdraw* method to the *CurrentAccount* but with an added fee for each withdrawal. This test functions in the same manner as the *CurrentWithdrawTest* above, only using the *SavingsAccount* version of the *withdraw* method. The code can be found in Appendix B.6.

#### CurrentDoubleWithdrawTest

Two withdrawals from a *CurrentAccount* are attempted in this test. To pass, only one of the withdrawals can be successful, otherwise the account balance will be below the limit set on the account. The *withdraw* methods makes multiple checks to the current balance before allowing a transaction to take place. However, in a thread-unsafe scenario, if one thread were to check the balance just before another thread made a successful withdrawal from the same account, then the balance it has will no longer be correct. The second thread will then be permitted to perform

a second withdrawal over the limit. Since the actions are identical, the execution order is not important. The code can be found in Appendix B.7.

### **CurrentWithdrawLoanDepositTest**

This test checks that the *LoanAccount* *deposit* method functions correctly with a corresponding *CurrentAccount*. The user takes out a loan so that the *CurrentAccount* initially has a positive balance and the *LoanAccount* has a negative balance. The *CurrentAccount* attempts to make a withdrawal that exceeds the limit and waits for funds to become available. Then a deposit will be made to the *LoanAccount*, such that the balance becomes positive and the excess is deposited into the *CurrentAccount* and the withdrawal from this account can proceed. For this test to pass, both methods must be functioning in a perfectly thread-safe way. The code can be found in Appendix B.8.

### **CurrentTransferTest**

The *transfer* method moves money from one *Account* to another by withdrawing from one and depositing in the other. In one scenario, the first thread will execute a transfer transaction and the second a deposit transaction. This is similar to the *CurrentWithdrawTest*, where the sender attempts to transfer too much money to another account and has to wait for available funds from a deposit. The other scenario is the reverse order in which the account will have enough money to perform a transfer. The code can be found in Appendix B.9.

### **CurrentTransferDepositWithdrawTest**

The final test combines all of the concurrent methods in three separate threads to maximise the chances of interference. Two *CurrentAccounts* are created, one sender and one receiver. The sender attempts to withdraw a large sum of money, transfer money to the other account and finally deposit a large enough sum to allow the other two transactions to be successful. The combination of the three methods produces up to six different scenarios depending on the order of the thread execution. The code can be found in Appendix B.10.

All of these tests determine a pass or failure by evaluating the balance of each *Account* object after all of the concurrent actions have taken place. The nature of the tests ensures that the outcome should always be the same for correct functionality but a mutant will likely produce a different outcome. The range of tests here covers all of the Banking system classes and all concurrent operations within them.

### 3.4.2 Incrementer Concurrent Tests

#### LockTest

The *LockTest* performs two *incrementLocked* methods in separate threads to produce the interference bug shown in Section 2.1.2 Table 1. Removing the locks through a mutation will mean that the threads are attempting to access and modify the *Incrementer* object's count at the same time. The code can be found in Appendix C.3.

#### InterruptTest

This tests the *incrementInterrupt* method by calling this in one thread and an *incrementLocked* in another. If the *interrupt* call has been removed by a mutation, then only one set of increments will be applied to the count and the final value will not be correct. The code can be found in Appendix C.4.

#### AwaitSignalTest

The *AwaitSignalTest* tests for the presence of the *await* and *signalAll* calls in the *incrementAwait* and *incrementSignal* methods. The functionality of the *incrementAwait* in this test, depends on the *incrementSignal* to be called and release it from its interrupted waiting state. The test will always pass with unaltered code, however it will not always fail when a mutation is introduced. This is because the outcome of this scenario depends on the order in which the two threads run. Thus, this test must be run multiple times until the mutation has been veritably caught. The code can be found in Appendix C.5.

Similar to the Banking system concurrent tests, the critical variable that is evaluated after concurrent actions have terminated is the count, in this case.

#### SyncTest

This test executes two *SyncIncRunnables* in separate threads with the intention of interference occurring during the increment operations. With synchronized code, the two threads will execute sequentially and will always produce the same outcome. With mutated code, the *synchronized* keyword modified to *this*, the code is no longer synchronized and will likely produce a different outcome every time. The code can be found in Appendix C.6

### 3.5 Software Engineering Process

IntelliJ was chosen as the IDE to develop in, utilising its intuitive layout, integration with GitHub and the various useful services it provides for refactoring and debugging<sup>5</sup>. Throughout development, GitHub was used to maintain version control and keep a record of daily updates. Initially, a logbook was created and updated regularly with project development ideas; resources found during research were recorded and any useful details kept for reference. This process helped provide a structure for the project and allowed for a continual review of the software design. Once development of the software was under way and the main architectural design had been outlined, the GitHub commit messages served a similar purpose. As well as describing any changes to the code, the following steps to take in development and any problems that had been encountered were documented here. The commit messages were reviewed upon revisiting the project as a reminder of the progress.

During the design and requirements gathering stages of development, initial research was conducted into concurrency, mutation testing and the relevant tools currently available for developers. Pitest was the most comprehensive of the tools discovered and a lot of the mutation tool's requirements were inspired by this software's features. Section 2.2 on Mutation Testing describes the similar aspects of PIT that have been implemented into this tool. The main difference between PIT and the mutation tool developed in this project, is the level at which the mutations are applied. PIT applies the operators on the byte code, whereas this mutation tool operates on the source code by manipulating strings.

Another approach to designing an automatic mutation tool, rather than utilising regular expressions to find and replace strings, is to use a parser tool. An existing grammar for the language, Java in this case, can be modified to recognise the relevant concurrent features and manipulate them according to the mutation operator specifications. Alternatively, a bespoke grammar could be built to only recognise the necessary features. This is likely a vastly more complex and time consuming method, but may benefit from being more general and can be easily applied to any system. Prior to developing the mutation tool using the regular expressions, the parser approach was considered using *ANTLR*<sup>6</sup>, a parser generator application, to modify an existing grammar for Java<sup>7</sup>. After some research, this method appeared to be too large a task for this project and it was rejected in

---

<sup>5</sup><https://www.jetbrains.com/idea/>

<sup>6</sup><https://www.antlr.org/>

<sup>7</sup><https://github.com/antlr/grammars-v4/blob/master/java8/Java8.g4>

favour of the simpler regular expressions mechanism.

An iterative approach was taken to developing the mutation tool; each mutation operator was implemented and tested before beginning development on the next one. To manage the work flow and to provide a structure to the development, the most pressing issues were displayed on a whiteboard and ticked off when completed. This is similar to the project management style found in an agile Kanban development with goal-oriented coding blocks, continuous revisions of the process and the use of Kanban cards for issue tracking (Radigan 2019). Organising the project in this manner promotes productivity and keeps development on schedule.

Aside from the concurrent unit tests written for the Banking and Incrementer systems, the mutation tool also required tests. Once a few of the mutation operators had been implemented into the tool, unit tests were created for the various methods within the Mutator class. However, as the project progressed, there became an unmanageable number of different options for each mutation operator and it was not an efficient use of time to create a unit test for each mutation. Sample files containing a small selection of the different mutable methods were created. Since many of the mutations operated on a shared mechanism of removal or parameter modification, the mutation tool was tested on only a few different methods and the outcomes were reasonably assumed to be representative of the whole selection. This method of manual observation is not necessarily as reliable as running a full suite of JUnit tests after any code modification, but the time saved during the development was worth the sacrifice.

Each file in the concurrent systems was stored outside of the IntelliJ project, isolated from the mutation testing tool. These files were intact and contained no errors or mutations; the mutation tool would accept these clean files as input before corrupting them with mutations. After applying the mutation, unit tests had to be performed on the new file, thus each mutant had to overwrite the previous mutation and then introduce the file to the IntelliJ project where the unit tests were stored. This method ensured that the original files remained intact and that only one mutation would be present in the project version. Simultaneously, each mutation was saved to another location on file to keep a record of results.

## 4 Analysis

### 4.1 Results

For a test to be considered successful at killing a mutation, the outcome of the test must go from passing when applied to clean, unaltered code, to failing when applied to mutated code. The test resulting in a failure indicates that the test has identified that the behaviour of the system has changed due to the mutation i.e. the test has fulfilled its purpose.

Since it can be difficult to predict the scheduling order of concurrently running threads, all concurrent tests were run up to 10 times or until a mutation was killed. This ensured that the tests covered all possible run scenarios to take place. If the test managed to catch a bug in at least one of the runs then it was deemed successful, regardless of whether other runs failed to kill a mutation. For each mutation, all of the tests relating to the mutated file were run and the result of a mutation surviving or not was assessed across the whole range of tests. If one of the tests reported the presence of a mutation but the others did not, the mutation would still be deemed to have been successfully killed.

The results of the tests will be displayed in sections 4.1.1 & 4.1.2, where green results indicate that a mutation was successfully killed i.e. the test returned with a failure. Red indicates that a mutation went undiscovered i.e. the test returned with a pass. Blue indicates that some other exception was thrown. Orange results indicate that the mutation produced a syntax error that prevented the tests from executing; these results will not be included in the mutation score as a pass or as a fail.

The JUnit tests have a slight variation in what is considered a test failure compared to the concurrent tests. JUnit tests can produce a failure when the observed state of the variables after execution are in accordance with the expected outcome, such as a correct account balance, but if an *IllegalMonitorStateException* is thrown, for example, then the test will not pass despite the methods otherwise functioning properly. In opposition, determining a failure for the concurrent tests relies purely on the outcome of the various account balances and exceptions are treated as blue results. The reason for the separate definitions is to differentiate between a failed test due to a behavioural change that caused the system to result in a different state to the expected, as opposed to an error thrown but the system still managed to result in the correct final balance. For all intents and purposes, a blue result can be considered to have killed a mutation as the error produced



indicates to the user that the system is not working perfectly. The two most common types of exceptions that are thrown from a mutation that will produce a blue result for concurrent tests, are the *IllegalMonitorStateException* and the *NullPointerException*.

**IllegalMonitorStateException (IMSE)** - API description: *"Thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor."* (Oracle 2018a). This typically occurs when removing a *lock* method whilst leaving the corresponding *unlock* method intact. The error is thrown because the command to release the object's lock on the monitor is given, even though the object never obtained the lock in the first place, due to the mutation.

**NullPointerException (NPE)** - API description: *"Thrown when an application attempts to use null in a case where an object is required."* (Oracle 2018b). A very familiar error to any Java developer, although, specifically in these tests, this error is almost exclusively thrown when removing a *newCondition*. Removing this method declaration in the *Incrementer* constructor consequently means that when the *withdraw* method attempts to call an *await* on the *fundsAvailableCondition*, the condition does not exist, hence the method is attempting to use null instead of an object.

A mutation test score will be provided for each results table and a combined score for the whole system. The mutation test score is a ratio of mutants killed to the total number of mutants tested, expressed as a percentage. One score will be given excluding blue results and one including them, the latter being the more significant value.

#### 4.1.1 Banking System Results

The results for all concurrent and JUnit tests for the Banking system are displayed in Tables 5-11.

JUnit Tests		Concurrent Feature					
Banking Concurrent Method		lock	unlock	interrupt	signalAll	await	newCondition
	Account constructor						
	Account deposit						
	Account transfer						
	Current withdraw						
	Savings withdraw						
	Loan deposit						

Table 5: JUnit test results for mutations applied to the selected concurrent methods in each of the Banking system classes. Mutation test score: 57%.

Account		Concurrent Feature			
Concurrent Test		lock	unlock	signalAll	newCondition
	Current Withdraw				
	Savings Withdraw				
	Current Transfer Deposit				
	Current Transfer Deposit Withdraw				
	Current Withdraw Loan Deposit				

Table 6: Results of concurrent tests for mutations applied to the *Account* class. Mutation test score excluding blue results: 50%. Score including blue results: 75%

Current		Concurrent Feature			
Concurrent Test		lock	unlock	interrupt	await
	Current Withdraw	blue	red	red	yellow
	Current Double Withdraw	blue	green	green	yellow
	Current Transfer Deposit	blue	red	red	yellow
	Current Transfer Deposit Withdraw	blue	green	red	yellow
	Current Withdraw Loan Deposit	green	red	red	yellow

Table 7: Results of concurrent tests for mutations applied to the *CurrentAccount* class. Mutation test score: 100%.

Savings		Concurrent Feature			
Concurrent Test		lock	unlock	interrupt	await
	Savings Withdraw	blue	red	red	yellow
	Savings Double Withdraw	blue	green	green	yellow

Table 8: Results of concurrent tests for mutations applied to the *SavingsAccount* class. Mutation test score excluding blue results: 75%. Score including blue results: 100%

Loan		Concurrent Feature	
Concurrent Test		lock	unlock
	Current Withdraw Loan Deposit	blue	red

Table 9: Results of concurrent tests for mutations applied to the *LoanAccount* class. Mutation test score excluding blue results: 0%. Score including blue results: 50%.

### Concurrent Tests

		Concurrent Feature					
		lock	unlock	interrupt	signalAll	await	newCondition
Banking Concurrent Method	Account constructor						
	Account deposit						
	Account transfer						
	Current withdraw						
	Savings withdraw						
	Loan deposit						

Table 10: Concurrent test results for mutations applied to the selected concurrent methods in each of the Banking system classes. Mutation test score excluding blue results: 57%. Score including blue results: 86%.

### Combined Results

		Concurrent Feature					
		lock	unlock	interrupt	signalAll	await	newCondition
Banking Concurrent Method	Account constructor						
	Account deposit						
	Account transfer						
	Current withdraw						
	Savings withdraw						
	Loan deposit						

Table 11: Combined test results for the Banking system JUnit tests and Concurrent Tests. Mutation test score: 86%.

#### 4.1.2 Incrementer System Results

The results for all JUnit and concurrent tests for the *Incrementer* system are displayed in Tables 12 & 13. Since the concurrent tests managed to kill all of the mutations, the combined results of the JUnit and concurrent tests are the same as just the concurrent test results.

Note that *SyncIncRunnable* is displayed as a method in Tables 12 & 13, where in fact it is a separate class that implements the *Runnable* interface to allow the *increment* method to be executed in synchronisation with multiple threads. The method that is actually mutated by the *MSP* operator, is the *run()* method, but *SyncIncRunnable* is displayed for better clarity.

##### JUnit Tests

		Concurrent Feature						
		lock	unlock	interrupt	signalAll	await	newCondition	synchronized
Incrementer Concurrent Method	Incrementer constructor							
	increment- Locked							
	increment- Interrupt							
	increment- Signal							
	increment- Await							
	SyncInc- Runnable							

Table 12: JUnit test results for the *Incrementer* system. Mutation score: 50%.

Concurrent Tests		Concurrent Feature						
Incrementer Concurrent Method		lock	unlock	interrupt	signalAll	await	newCondition	synchronized
	Incrementer constructor							
	increment- Locked							
	increment- Interrupt							
	increment- Signal							
	increment- Await							
	SyncInc- Runnable							

Table 13: Test results for *Incrementer* system concurrent tests. Mutation score: 100%.

## 4.2 Results Analysis

The most surprising finding from the results is how well the JUnit tests performed at identifying mutations without having to run multiple threads. Comparing the mutation test scores of the JUnit tests with the concurrent tests, the JUnit tests matched the concurrent ones for the scores excluding blue results with 57%, respectively found in Tables 5 & 10. However, when including the blue results, the concurrent tests come out on top with 86%.

The concurrent tests have proven to be very effective when including the errors from the blue results, reaching a mutation score of 86% which matches that of the combined results with the JUnit tests. This could suggest that the JUnit tests are superfluous, as they do not kill any additional mutants to the concurrent ones. Although, from a development point of view, JUnit tests are quick and easy to produce and can provide instant feedback for many concurrency bugs in a system without having to design bespoke tests for each concurrency issue that can occur. JUnit tests are also extremely useful for ensuring that the basic functionality of methods function in the expected manner, whereas the concurrent tests are more representative of the actual use of the system and confirm the robustness of handling concurrent issues. The ideal testing suite would utilise JUnit and concurrent tests to reap the benefits of both styles.

Compared to the *Banking* system, the *Incrementer* system JUnit tests have been slightly less successful reporting a lower mutation score of 50% to the 57% found in the *Banking* system (Tables 5 & 12). The concurrent tests, however, were extremely effective at killing mutations and significantly outperformed the *Banking* system: 57% excluding blue results and 86% including them compared to 100% in the *Incrementer* system, respectively (Tables 10 & 13). Finally, the combined results also noted an improvement from the *Banking* system: 86% to 100% (Tables 11 & 13). Initially this may seem surprising that both style of tests did not follow the same trend of either being more or less successful with respect to the *Banking* system. However, this is likely due to the overall design of the two systems serving different purposes. The *Banking* system was designed to be a useful piece of software and had specific functional requirements to meet. It is larger and more complex than the *Incrementer* system, with multiple different interacting methods across a range of classes. On the other hand, the *Incrementer* system functionality was not designed to serve a useful purpose to the user. Instead, the methods were designed with a focus on the concurrent tests and discovering the most simple procedures that could be tested with ease. The complexity of the *Banking* system consequently means that various methods are co-dependent on each other and more noticeable problems likely to arise when introducing a mutation. This produced small domino effects on the dependent methods that allowed the JUnit tests a greater opportunity to identify faults in the code, without having to utilise multiple threads. In contrast, the *Incrementer* system's simple design with independent methods means that altering concurrent features has little to no effect on single threaded execution, but dramatic effects on multi-threaded execution.

For the case of the *await* statements in the *CurrentAccount* and *SavingsAccount withdraw* methods and the *Incrementer* system *incrementAwait* method, any mutation applied to it produced an *InterruptedException* error that prohibited compilation. The *await* method must always be accompanied by an *InterruptedException*, which will be thrown if the method is removed unless there are multiple *await* calls in the code segment. Consequently, no test was able to run after this mutation, although the mutation can be considered to have been killed as the user would become aware of the error when attempting to compile the code. Comparing to the Pitest tool's definitions in section 2.2, this would be considered a 'run error' result and cannot be considered in the mutation test scores as a result.

The *RCXC* mutation operator removing the *lock* method, always produced an *IllegalMonitorStateException*. JUnit tests flag this as a failure, and although the concurrent tests reported the same exception, executing the code produced the same outcome as the pre-mutated version of the code, so it is reported as a blue result in Tables 6-10. Relating back to the concurrent bug patterns identified by

Farchi et al. (2003), this is an example of a "Wrong Lock or No Lock Bug Pattern" found in section 2.4.1. The *lock* method has been removed and so the section of code does not acquire the lock on thread execution, leaving the code unprotected. The *IMSE* exception is thrown because the *unlock* method remains in the code segment, attempting to release a lock that has been acquired.

When catching the mutations that removed the *unlock* and *interrupt* methods, executing the code would cause the system to hang and the threads would never achieve termination. For the *unlock*, this occurs because whichever thread executes first and obtains the lock on the monitor would never release the lock to allow the other thread to begin execution. Referring to bug patterns identified by Bradbury et al. (2006), this is an example of a 'blocking critical section' bug, which is defined in section 2.4.3. If the system hangs when running a test and cannot complete execution to provide a result, then the test has in effect failed and the mutation can be considered to be caught. When this has occurred, these results have been marked as green in the various results tables.

For the removal of the *fundsAvailableCondition* instantiation with *newCondition* in the *Account* class and the matching case in the *Incrementer* system, a *NullPointerException* was thrown, which resulted in the blue results seen in Table 6. However, the *CurrentWithdrawTest* did manage to kill the mutation by observing an unexpected account balance value and is consequently marked as a green result. Similarly, the *AwaitSignalTest* in the *Incrementer* system reports a positive mutation catch with a count different to the expected value.

### 4.3 Methodology Analysis

The *Incrementer* system has been designed to purposefully encounter concurrent problems, such as thread interference and memory consistency errors. The reason behind this is to easily catch the bugs that are present after a mutation has taken place in one of the system class files. *Sleep()* statements were utilised to slow down the computation time and increase the likelihood of two threads attempting to access state variables within the same window. This serves as a proof of concept that it is possible and fairly straightforward to design effective unit tests for concurrent code. Although, obviously, in real-life scenarios, compromising the performance and execution time of large systems is not ideal. Especially considering the main motivation of implementing concurrency to a system is to improve the performance. The test environment can be used to determine whether the system has been properly safeguarded with less concern for efficiency. There may be an



argument to introduce a default sleep time of 0 seconds, stored as a variable, to be used as a parameter for the relevant methods, so that in normal use the system will function with the desired efficiency. However, when it comes to testing, this variable can be altered to a more suitable value that will provide significant opportunity for errors to arise that would remain hidden in the default state of execution.

It was deemed sufficient to test the systems with only two threads in most cases. Since the systems were small in scale and designed in such a manner that errors would be readily caught, increasing the number of threads would be superfluous. If a mutation was identified and killed by any unit test with only two interleaving threads, then the mutation would definitely be caught in number of more threads. However, in larger, more complex systems, any bug may be significantly harder to find, so increasing the number of threads with the objective of maximising the likelihood of interference could be an effective approach. Also, running more than two threads concurrently could reduce the time spent testing a system as test would not necessarily have to be run nearly as many times to confidently convince the user that all possible scenarios have taken place. The Executor Service mentioned in Section 2.1.5 provides the functionality to efficiently manage many threads at once.

In terms of refactoring, it might have been prescient to break the structure of the tool down into separate subclasses for each mutator. In this case, the class hierarchy would have an abstract Mutator parent class with each mutator inheriting from it. This would have the positive effect of expandability when introducing new mutation operators. The core methods could be inherited with ease and only the necessary methods overridden. This would follow the Object Oriented paradigm to greater effect. One drawback, however, is that a runnable class would have to be created for each mutator, whereas with just one mutator class, the user can choose a different operator with ease when performing multiple tests.

It may be noted that there are more concurrent unit tests for the *Banking* system than the *Incrementer* system. The reasons for this variance are partly due to the fact that the *Banking* System is larger and more complex; the Banking System consists of four different types of *Account* with many more methods that can interact with the same resources during concurrent execution. Some of the tests for the *Incrementer* system also manage to kill multiple mutations due to the simplicity of the system and because it was designed with the purpose of being easily tested. To test the same number of mutations, half as many concurrent tests were applied in the *Incrementer* system and an additional test was created for the

*MSP* mutation of the *synchronized* keyword.

With the analysis of the results above, some of the concurrent tests for the *Banking* system could be refined and improved upon to consistently catch bugs relating to removing the locks and the *await/signalAll* methods. The *Incrementer* system tests prove that it is possible to kill all of these mutations. The success of the concurrent tests proves that it is entirely possible to design a suite of effective unit tests for concurrent software. Although, the results from the concurrent tests in the *Banking* system suggests that it is more difficult than producing unit tests for single threaded software. A solution could not be found to successfully kill mutations involving the *signalAll* method in the *Account* class or the *unlock* method in the *Loan* class. Expanding the selection of the concurrent tests or increasing the number of concurrently executing threads may achieve the desired result, although the results from the *Incrementer* system prove that it is possible to detect these mutations in just two threads. Ultimately, it may depend on the precise use of the concurrent features in the different contexts.

## 5 Recommendations

The original goal was to end up with a full test suite that would seed many mutations throughout the code, perform the unit tests and produce a set of results. The results would detail which mutations were applied, identify which tests managed to kill mutations and produce the ratio of successfully caught mutations to the total number of seeded errors. Due to time limitations, a revision of expectations and priorities, the focus was on developing the mutation system and the unit tests for the sample test systems. Thus, many of the stages above were carried out manually. The inspiration for the tool described was spawned from the presence of other mutation testing tools, such as Pitest, which provides a full mutation testing environment and many operators to select. However, none of the available tools provide support for concurrent operators due to the special difficulties that surround the subject. A complete automatic tool for evaluating the effectiveness of concurrent tests would be a valued area for future development.

Although the concurrent tests were effective in this context, this may not be the case in other scenarios, especially larger and more complex systems. With more time and the availability of more legitimate pre-existing concurrent systems, it would be pertinent to apply the mutation operators explored in this project to these systems and verify the results obtained here. This would give a greater insight into the effectiveness unit tests against concurrent issues.

As well as testing a greater range of concurrent systems, there are many more mutation operators that could be developed for this tool. The automatic mutation application tool currently only supports application of four mutation operators with all of the relevant concurrent keywords and methods: *MXT*, *MSP*, *RTXC* and *RCXC*. However, at least 7 more of the operators could be implemented with little difficulty due their similar mechanisms. These other operators can be found in Appendix A Table 14 along with the full list of operators defined by Bradbury et al. (2006). The regular expression used is very consistent for simple removals of keywords and minor modifications to method parameters. The operators that have been implemented already were chosen based on the needs of the banking system and the limited number of concurrent methods present in the code. Expanding the selection of mutation operators would allow for greater flexibility in choosing concurrent systems and provide a more thorough analysis.

For a larger project or a publicly available application, parser manipulation is potentially a more robust and general purpose method of applying mutations and would be more suited to these types of project. This area could be researched

more thoroughly to develop a more complete concurrency mutation tool that aims to implement more of the mutation operators by Bradbury et al. (2006).

IntelliJ provides code coverage statistics that report the number of lines and branches that are covered by a test suite (JetBrains 2019). A similar idea could be applied to concurrent tests, for instance, to prevent *lock* and *unlock* methods from producing "Wrong Lock or No Lock" and "Blocking Critical Section Code" bug patterns (Sections 2.4.1 & 2.4.3), a simple tool could be created that counts the number of instances of each of these method calls; there should be an equal number of these calls. If the counts differ then the programmer would be informed and all instances will be returned to the user to identify where the missing call/s are located. Similar mechanisms could be implemented for other paired methods, such as *await* and *signalAll*.

## 6 Conclusion

The objectives of this project were clearly defined: to produce an automatic mutation tool, apply mutations to concurrent software and evaluate the effectiveness of the systems unit tests. These aims have been successfully met, the test results have been evaluated with reflections on improvements in the unit tests and suggestions have been made for future work in this area.

The topics of concurrency and mutation testing were covered in sufficient detail to provide the necessary understanding for this project. With this knowledge and reviewing the relevant literature, particularly focusing on the works of Farchi et al. (2003) and Bradbury et al. (2006), a proof-of-concept mutation tool was developed with four mutation operators implemented: the *MXT*, *RTXC*, *RCXC* and *MSP* operators can mutate a wide variety of different concurrent features through modification or removal. The potential for implementing more is easily attainable with more time, as a number of other operators share similar mutation mechanisms.

The tool was tested on two different systems with a range of concurrent features and the tests for these systems have been evaluated based on the results. A mutation score has been given in each example to evaluate the effectiveness of the tests and the results tables indicate the areas that need improvements. The first of the software tested, the *Banking* system received an overall mutation score of 86% after seeding a total of 14 different mutations, combining the efforts of single-threaded JUnit tests and multi-threaded concurrent tests. The second system to be tested, the *Incrementer* system, was created specifically for this project and was designed to be easily tested for concurrent bugs, received a mutation score of 100% for the 12 mutations applied. These scores represent the percentage of mutations that were detected and successfully killed compared to the total number that were seeded.

Taking the results presented throughout this report into consideration and the techniques employed during development, design patterns can be created to aid programmers in writing thread-safe code and effective unit tests to ensure this. Particular concurrent features to focus on are the *lock*, *unlock*, *await* and *signalAll* methods as they frequently avoided detection by the suite of unit tests. Of the seven concurrent features explored in this project, all but one were successfully tested and the mutation killed by at least one test. Aspects of these successful concurrent tests can be applied to the design of tests for other concurrent features and operators discussed in the work of Bradbury et al. (2006).

The purpose of JUnit tests is most often to provide a satisfactory line and branch coverage for a system, ensuring that each code segment is accounted for with an expected outcome. This style of testing does not lend itself well to concurrent software which is inherently unpredictable with its thread execution order mostly outwith the programmer's control. The concurrent tests were designed to target these concurrency issues by creating a variety of scenarios that accentuate situations involving interleaving threads. Reporting such high mutation scores serves as proof that it is entirely possible to design effective unit tests for concurrent software.

# Bibliography

Adrion, W. R., Branstad, M. A. and Cherniavsky, J. C. (1981), *Validation, verification, and testing of computer software*, U.S. Dept. of Commerce, National Bureau of Standards.

Ammann, P. and Offutt, J. (2017), *Introduction to software testing*, Cambridge University Press.

Bradbury, J. S., Cordy, J. R. and Dingel, J. (2006), ‘Mutation operators for concurrent java (j2se 5.0)’, *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)* .

Coles, H. (2019), ‘Real world mutation testing - mutators’.

**URL:** <http://pitest.org/quickstart/mutators/>

Farchi, E., Nir, Y. and Ur, S. (2003), ‘Concurrent bug patterns and how to test them’, *Proceedings International Parallel and Distributed Processing Symposium* p. 286.

Gamma, E., Helm, R., Johnson, R. E. and Vlissides, J. (2015), *Design patterns: elements of reusable object-oriented software*, Pearson Education.

JetBrains (2019), ‘Code coverage’.

**URL:** <https://www.jetbrains.com/help/idea/code-coverage.html>

Mois, M. (2015), ‘Java concurrency essentials’.

Oracle (2017), ‘Lesson: Regular expressions’.

**URL:** <https://docs.oracle.com/javase/tutorial/essential/regex/index.html>

Oracle (2018a), ‘Class illegalmonitorstateexception’.

**URL:** <https://docs.oracle.com/javase/7/docs/api/java/lang/IllegalMonitorStateException.html>

Oracle (2018b), ‘Class nullpointerexception’.

**URL:** <https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html>

Oracle (2019a), ‘Lesson: Concurrency’.

**URL:** <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

Oracle (2019b), ‘Package java.util.regex’.

**URL:** <https://docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html>

Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D. and Holmes, D. (2005), *Java Concurrency in Practice*, Addison-Wesley Professional.

Radigan, D. (2019), 'Kanban - a brief introduction'.

**URL:** <https://www.atlassian.com/agile/kanban>

Silberschatz, A., Galvin, P. B. and Gagne, G. (2013), *Operating System Concepts*, 9th edn, Wiley Publishing.



## 7 Appendices

### A Mutation Operators

Operator Category	Concurrency Mutation Operators for Java (J2SE 5.0)
Modify Parameters of Concurrent Methods	<b>MXT</b> – Modify Method-X Time ( <i>wait()</i> , <i>sleep()</i> , <i>join()</i> , and <i>await()</i> method calls)
	<b>MSP</b> - Modify Synchronized Block Parameter
	<b>ESP</b> - Exchange Synchronized Block Parameters
	<b>MSF</b> - Modify Semaphore Fairness
	<b>MXC</b> - Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	<b>MBR</b> - Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls	<b>RTXC</b> – Remove Thread Method-X Call ( <i>wait()</i> , <i>join()</i> , <i>sleep()</i> , <i>yield()</i> , <i>notify()</i> , <i>notifyAll()</i> Methods)
	<b>RCXC</b> – Remove Concurrency Mechanism Method-X Call ( <i>methods in Locks, Semaphores, Latches, Barriers, etc.</i> )
	<b>RNA</b> - Replace <i>notifyAll()</i> with <i>notify()</i>
	<b>RJS</b> - Replace <i>Join()</i> with <i>Sleep()</i>
	<b>ELPA</b> - Exchange Lock/Permit Acquisition
	<b>EAN</b> - Exchange Atomic Call with Non-Atomic
Modify Keyword	<b>ASTK</b> – Add Static Keyword to Method
	<b>RSTK</b> – Remove Static Keyword from Method
	<b>RSK</b> - Remove Synchronized Keyword from Method
	<b>RSB</b> - Remove Synchronized Block
	<b>RVK</b> - Remove Volatile Keyword
	<b>RFU</b> - Remove Finally Around Unlock
Switch Concurrent Objects	<b>RXO</b> - Replace One Concurrency Mechanism-X with Another ( <i>Locks, Semaphores, etc.</i> )
	<b>EELo</b> - Exchange Explicit Lock Objects
Modify Critical Region	<b>SHCR</b> - Shift Critical Region
	<b>SKCR</b> - Shrink Critical Region
	<b>EXCR</b> – Expand Critical Region
	<b>SPCR</b> - Split Critical Region

Table 14: List of concurrency mutation operators devised by Bradbury et al. (2006) The green highlighted operators have been fully implemented in the mutation testing tool. The blue highlighted operators could be implemented without difficulty.

Java (J2SE 5.0) Concurrency Mutation Operator Categories	Threads	Synchronization methods	Synchronization statements	Synchronization with implicit monitor locks	Explicit locks	Semaphores	Barriers	Latches	Exchangers	Built-in concurrent data structures (e.g. queues)	Built-in thread pools	Atomic variables (e.g. LongInteger)
<i>Modify Parameters of Concurrent Methods</i>	✓	–	✓	✓	✓	✓	✓	✓	–	–	–	–
<i>Modify the Occurrence of Concurrency Method Calls</i>	✓	–	–	–	✓	✓	✓	✓	–	–	–	✓
<i>Modify Keyword</i>	–	✓	✓	✓	✓	–	–	–	–	–	–	–
<i>Switch Concurrent Objects</i>	–	–	–	–	✓	✓	✓	✓	✓	✓	✓	–
<i>Modify Concurrent Region</i>	–	✓	✓	✓	✓	✓	–	–	–	–	–	–

Table 15: The relationship between new mutation operators for concurrency and the concurrency features provided by J2SE 5.0 (Farchi et al. 2003).

## B Banking System Code

The four *Banking* system *Account* classes were taken from a Strathclyde undergraduate Computer Science class on concurrency and remain unaltered. The concurrent unit tests were written by myself for the purpose of evaluating the software.

### B.1 Account Class

```

1 package bankSystem;
2 import java.util.concurrent.locks.Condition;
3 import java.util.concurrent.locks.ReentrantLock;
4
5 public abstract class Account {
6
7     protected static final int TIMEOUT = 5;
8     protected double balance;
9     protected int accountNumber;
10    protected String name;
11    protected ReentrantLock balanceLock;
12    protected Condition fundsAvailableCondition;
13
14    public Account(String name, double balance) {
15        this.balance = balance;

```

```

16     this.name = name;
17     System.out.println("Initial balance of " + name + " is " + balance
18         + ".");
19     balanceLock = new ReentrantLock();
20     fundsAvailableCondition = balanceLock.newCondition();
21 }
22
23 public int getAccountNumber() {
24     return this.accountNumber;
25 }
26
27 public double getBalance() {
28     System.out.println("Balance of " + name + " is " + balance + ".");
29     return this.balance;
30 }
31
32 public void deposit(double amount) {
33     System.out.println("Depositing " + amount + " in " + name + "...");
34     balanceLock.lock();
35     try {
36         this.balance += amount;
37         getBalance();
38         fundsAvailableCondition.signalAll();
39     } finally {
40         balanceLock.unlock();
41     }
42 }
43
44 public abstract boolean withdraw(double amount);
45
46 public void transferMoney(double amount, Account recipient) {
47     System.out.println("Transferring " + amount + " from " + name + "
48         to " + recipient.getName() + ":");
49     balanceLock.lock();
50     try {
51         if (withdraw(amount))
52             recipient.deposit(amount);
53         getBalance();
54         recipient.getBalance();
55     } finally {
56         balanceLock.unlock();
57     }
58 }

```

```

57
58     public String getName() {
59         return name;
60     }
61 }

```

## B.2 CurrentAccount Class

```

1  package bankSystem;
2  import java.util.concurrent.TimeUnit;
3
4  public class CurrentAccount extends Account {
5
6      private final double overdraftLimit;
7
8      public CurrentAccount(String name, double initialBalance) {
9          super(name, initialBalance);
10         this.overdraftLimit = -50;
11     }
12
13     @Override
14     public boolean withdraw(double amount) {
15         System.out.println("Withdrawing " + amount + " from " + name +
16             "...");
17         boolean waiting = true;
18         balanceLock.lock();
19         try {
20             while (balance - amount < overdraftLimit) {
21                 if (!waiting) {
22                     Thread.currentThread().interrupt();
23                 } else {
24                     System.out.println("Waiting for funds to become
25                         available...");
26                 }
27                 waiting = fundsAvailableCondition.await(TIMEOUT,
28                     TimeUnit.SECONDS);
29             }
30         }
31
32         this.balance -= amount;
33         getBalance();
34         return true;
35     }
36 }

```

```

31
32     } catch (InterruptedException e) {
33         System.out.println("Can't withdraw " + amount + " from account
34             with balance " + balance);
35         return false;
36     } finally {
37         balanceLock.unlock();
38     }
39
40     public double getOverdraftLimit() {
41         return overdraftLimit;
42     }
43 }

```

### B.3 SavingsAccount Class

```

1 package bankSystem;
2 import java.util.concurrent.TimeUnit;
3
4 public class SavingsAccount extends Account {
5
6     private double fee;
7
8     public SavingsAccount(String name, double initialBalance) {
9         super(name, initialBalance);
10        this.fee = 0.5;
11    }
12
13    @Override
14    public boolean withdraw(double amount) {
15        System.out.println("Withdrawing " + amount + " from " + name +
16            "...");
17        boolean waiting = true;
18        balanceLock.lock();
19        try {
20            while (balance - (amount + fee) < 0) {
21                if (!waiting) {
22                    Thread.currentThread().interrupt();
23                } else {

```

```

23         System.out.println("Waiting for funds to become
24             available...");
25     }
26     waiting = fundsAvailableCondition.await(TIMEOUT,
27         TimeUnit.SECONDS);
28 }
29 this.balance -= (amount + fee);
30 getBalance();
31 return true;
32
33 } catch (InterruptedException e) {
34
35     System.out.println("Can't withdraw " + amount + " from account
36         with balance " + balance);
37     return false;
38 } finally {
39     balanceLock.unlock();
40 }
41 }
42 }

```

## B.4 LoanAccount Class

```

1 package bankSystem;
2
3 public class LoanAccount extends Account {
4
5     private CurrentAccount parentAccount;
6
7     public LoanAccount(String name, double loanAmount, CurrentAccount
8         parent) {
9         super(name, -loanAmount);
10        this.parentAccount = parent;
11        parentAccount.deposit(loanAmount);
12    }
13
14    @Override
15    public boolean withdraw(double amount) {
16        System.out.println("Can't withdraw " + amount + ". Withdrawals not
17            available for loan account.");
18    }
19 }

```

```

16         return false;
17     }
18
19     @Override
20     public void deposit(double amount) {
21         balanceLock.lock();
22         try {
23             this.balance += amount;
24             if (balance >= 0) {
25                 System.out.println("Loan paid off. You have " + balance + "
                                     credit.");
26             }
27             if (balance > 0) {
28                 parentAccount.deposit(balance);
29                 balance = 0;
30                 getBalance();
31             }
32         } finally {
33             balanceLock.unlock();
34         }
35     }
36
37 }

```

## B.5 CurrentWithdrawTest Class

```

1 package bankSystemUnitTests;
2
3 import bankSystem.CurrentAccount;
4 import bankSystem.DepositRunnable;
5 import bankSystem.WithdrawRunnable;
6
7 public class CurrentWithdrawTest {
8     /**
9      * Unit test for CurrentAccount withdraw method
10     * Runs withdraw in one separate thread and deposit in another
11     * @param args
12     * @throws InterruptedException
13     */
14
15     public static void main(String[] args) throws InterruptedException {

```

```

16     CurrentAccount account = new CurrentAccount("Account A (Current)",
17         0);
18
19     Thread thread1 = new Thread(new WithdrawRunnable(account, 800));
20     Thread thread2 = new Thread(new DepositRunnable(account, 1000));
21
22     thread1.start();
23     thread2.start();
24
25     thread1.join(3000);
26     thread2.join(3000);
27     assert account.getBalance() == 200: "Incorrect Balance. Balance: "
28         + account.getBalance() + " Expected Balance: 200.00";
29 }

```

## B.6 SavingsWithdrawTest Class

```

1 package bankSystemUnitTests;
2
3 import bankSystem.DepositRunnable;
4 import bankSystem.SavingsAccount;
5 import bankSystem.WithdrawRunnable;
6
7 public class SavingsWithdrawTest {
8     /**
9      * Unit test for SavingsAccount withdraw method
10     * Runs withdraw in one separate thread and deposit in another
11     * @param args
12     * @throws InterruptedException
13     */
14
15     public static void main(String[] args) throws InterruptedException {
16         SavingsAccount account = new SavingsAccount("Account A
17             (Current)", 0);
18
19         Thread thread1 = new Thread(new WithdrawRunnable(account, 800));
20         Thread thread2 = new Thread(new DepositRunnable(account, 1000));
21
22         thread1.start();

```



```

22         thread2.start();
23
24         thread1.join(3000);
25         thread2.join(3000);
26         assert account.getBalance() == 199.5: "Incorrect Balance.
           Balance: " + account.getBalance() + " Expected Balance:
           199.50";
27     }
28 }

```

## B.7 CurrentDoubleWithdrawTest Class

```

1  package bankSystemUnitTests;
2
3  import bankSystem.CurrentAccount;
4  import bankSystem.WithdrawRunnable;
5
6  public class CurrentDoubleWithdrawTest {
7      /**
8       * Unit test for CurrentAccount withdraw method
9       * Runs withdraw in two separate threads
10      * @param args
11      * @throws InterruptedException
12      */
13
14      public static void main(String[] args) throws InterruptedException {
15          CurrentAccount account = new CurrentAccount("Account A
              (Current)", 0);
16
17          Thread thread1 = new Thread(new WithdrawRunnable(account, 40));
18          Thread thread2 = new Thread(new WithdrawRunnable(account, 20));
19
20          thread1.start();
21          thread2.start();
22
23          thread1.join(3000);
24          thread2.join(3000);
25
26          assert (account.getBalance() == -40) | (account.getBalance() ==
              -20): "Incorrect Balance. Balance: " + account.getBalance() +
              " Expected Balance: -40.00 or -20.00";

```

```
27     }
28 }
```

## B.8 CurrentWithdrawLoanDepositTest Class

```
1 package bankSystemUnitTests;
2
3 import bankSystem.CurrentAccount;
4 import bankSystem.DepositRunnable;
5 import bankSystem.LoanAccount;
6 import bankSystem.WithdrawRunnable;
7
8 public class CurrentWithdrawLoanDepositTest {
9     /**
10      * Unit test for CurrentAccount withdraw method and LoanAccount
11      * deposit method
12      * Runs withdraw in one thread and deposit in another
13      * @param args
14      * @throws InterruptedException
15      */
16     public static void main(String[] args) throws InterruptedException {
17         CurrentAccount account = new CurrentAccount("Current Account",
18             0);
19         LoanAccount loanAccount = new LoanAccount("Loan Account", 100,
20             account);
21
22         assert account.getBalance() == 100: "Incorrect Balance. Balance:
23             " + account.getBalance() + " Expected Balance: 100.00";
24         assert loanAccount.getBalance() == -100: "Incorrect Balance.
25             Balance: " + account.getBalance() + " Expected Balance:
26             -100.00";
27
28         Thread thread1 = new Thread(new WithdrawRunnable(account, 800));
29         Thread thread2 = new Thread(new DepositRunnable(loanAccount,
30             1000));
31
32         thread1.start();
33         thread2.start();
34
35         thread1.join(3000);
36     }
37 }
```

```

30     thread2.join(3000);
31
32     assert account.getBalance() == 200: "Incorrect Balance. Balance:
33         " + account.getBalance() + " Expected Balance: 200.00";
34     assert loanAccount.getBalance() == 0: "Incorrect Balance.
35         Balance: " + account.getBalance() + " Expected Balance:
36         300.00";
37 }
38 }

```

## B.9 CurrentTransferTest

```

1  package bankSystemUnitTests;
2
3  import bankSystem.CurrentAccount;
4  import bankSystem.DepositRunnable;
5  import bankSystem.TransferRunnable;
6
7  public class CurrentTransferTest {
8      /**
9       * Unit test for CurrentAccount transfer method
10      * Runs transfer in one thread and deposit in another
11      * @param args
12      * @throws InterruptedException
13      */
14
15     public static void main(String[] args) throws InterruptedException {
16         CurrentAccount sender = new CurrentAccount("Current Account
17             Sender", 0);
18         CurrentAccount receiver = new CurrentAccount("Current Account
19             Receiver", 0);
20
21         Thread thread1 = new Thread(new TransferRunnable(sender,
22             receiver, 100));
23         Thread thread2 = new Thread(new DepositRunnable(sender, 100));
24
25         thread1.start();
26         thread2.start();
27
28         thread1.join(3000);
29         thread2.join(3000);
30     }
31 }

```

```

27
28     assert sender.getBalance() == 0: "Incorrect Balance. Balance: "
29         + sender.getBalance() + " Expected Balance: 0.00";
30
31     assert receiver.getBalance() == 100: "Incorrect Balance.
32         Balance: " + receiver.getBalance() + " Expected Balance:
33         100.00";
34
35 }
36
37 }

```

## B.10 CurrentTransferDepositWithdrawTest Class

```

1  package bankSystemUnitTests;
2
3  import bankSystem.CurrentAccount;
4  import bankSystem.DepositRunnable;
5  import bankSystem.TransferRunnable;
6  import bankSystem.WithdrawRunnable;
7
8  public class CurrentTransferDepositWithdrawTest {
9      /**
10       * Unit test for CurrentAccount transfer, deposit and withdraw method
11       * Runs transfer, deposit and withdraw in separate threads
12       * @param args
13       * @throws InterruptedException
14       */
15
16     public static void main(String[] args) throws InterruptedException {
17         CurrentAccount sender = new CurrentAccount("Account A
18             (Current)", 0);
19         CurrentAccount receiver = new CurrentAccount("Account B
20             (Current)", 0);
21
22         Thread thread1 = new Thread(new WithdrawRunnable(sender, 800));
23         Thread thread2 = new Thread(new TransferRunnable(sender,
24             receiver, 100));
25         Thread thread3 = new Thread(new DepositRunnable(sender, 1000));
26
27         thread1.start();
28         thread2.start();
29         thread3.start();
30     }
31 }

```

```

27
28     thread1.join(3000);
29     thread2.join(3000);
30     thread3.join(3000);
31     assert sender.getBalance() == 100 : "Incorrect Balance. Balance:
    " + sender.getBalance() + " Expected Balance: 100.00";
32     assert receiver.getBalance() == 100 : "Incorrect Balance.
    Balance: " + receiver.getBalance() + " Expected Balance:
    100.00";
33 }
34 }

```

## C Incrementer System Code

All *Incrementer* classes were written by myself for this project.

### C.1 Incrementer Class

```

1 package concurrentSystems;
2
3 import java.util.concurrent.TimeUnit;
4 import java.util.concurrent.locks.Condition;
5 import java.util.concurrent.locks.ReentrantLock;
6
7
8 public class Incrementer {
9     /**
10      * Incrementer class provides multiple concurrent methods that
11      * utilise the increment operation, x++.
12      */
13     private ReentrantLock lock = new ReentrantLock();
14     private int count;
15     private int secs;
16     private int millis;
17     private Condition condition;
18
19     /**
20      * Constructor

```

```

20     */
21     public Incrementer() {
22         count = 0;
23         secs = 3;
24         millis = 750;
25         condition = lock.newCondition();
26     }
27
28     /**
29     * Basic incrementer method, increases count by input value inc
30     * @param inc increments count
31     */
32     public void increment(int inc) {
33         if (inc > 0) {
34             System.out.println("Incrementing count by " + inc);
35             for (int i = 0; i < inc; i++) {
36                 count++;
37                 System.out.println(count);
38
39                 try {
40                     Thread.sleep(millis);
41                 } catch (Exception e) {
42                     System.out.println("Thread Interrupted");
43                 }
44             }
45         } else {
46             System.out.println("Increment amount must be greater than
47                 1");
48         }
49     }
50
51     /**
52     * Implements a lock system to the basic increment
53     * @param inc increments count
54     */
55     public void incrementLocked(int inc) {
56         System.out.println("incrementLocked begins...");
57         lock.lock();
58         increment(inc);
59         lock.unlock();
60         System.out.println("incrementLocked ends.");
61     }

```

```

62  /**
63   * If the current value of count %5 == 0 then the count will be
        incremented by inc. Otherwise count will be set to -1
64   * @param inc increments count
65   */
66  public void incrementInterrupt(int inc) {
67      System.out.println("incrementInterrupt begins...");
68      lock.lock();
69      if(count%5 == 0) {
70          Thread.currentThread().interrupt();
71      }
72
73      if(Thread.interrupted()) {
74          increment(inc);
75      } else {
76          count = -1;
77          System.out.println(count);
78      }
79
80      lock.unlock();
81      System.out.println("incrementInterrupt ends.");
82  }
83
84  /**
85   * Increments count if a signalAll is called in another thread or if
        the count != 0
86   * @param inc increments count
87   */
88  public void incrementAwait(int inc) {
89      System.out.println("incrementAwait begins...");
90      boolean waiting = true;
91      lock.lock();
92      try {
93          if (count == 0) {
94              System.out.println("Waiting");
95              waiting = condition.await(secs, TimeUnit.SECONDS);
96              if (waiting) {
97                  increment(inc);
98              }
99          } else {
100              increment(inc);
101          }
102

```

```

103     } catch (InterruptedException e) {
104         System.out.println("Thread interrupted.");
105
106     } finally {
107         lock.unlock();
108     }
109
110 }
111
112 /**
113  * Increments the count and calls a signalAll
114  * @param inc increments count
115  */
116 public void incrementSignal(int inc) {
117     System.out.println("incrementSignal begins...");
118     lock.lock();
119     increment(inc);
120     condition.signalAll();
121     System.out.println("incrementSignal ends.");
122     lock.unlock();
123 }
124
125 /**
126  * Getter for count
127  * @return count
128  */
129 public int getCount() {
130     return count;
131 }
132
133 /**
134  * Setter for count
135  * @param i new count value
136  */
137 public void setCount(int i) {
138     count = i;
139 }
140
141 /**
142  * Sets secs
143  * @param s new secs value
144  */
145 public void setSecs(int s) {

```



```

146     secs = s;
147 }
148
149 /**
150  * Sets millis
151  * @param m new millis value
152  */
153 public void setMillis(int m) {
154     millis = m;
155 }
156 }

```

## C.2 SyncIncRunnable Class

```

1 package concurrentSystems;
2
3 public class SyncIncRunnable extends Thread {
4     private Incrementer syncInc;
5     private int inc;
6
7     public SyncIncRunnable(Incrementer i, int x) {
8         syncInc = i;
9         inc = x;
10    }
11
12    public void run() {
13        synchronized(this) { //synchronized was modified
14            syncInc.increment(inc);
15        }
16    }
17 }

```

## C.3 LockTest Class

```

1 package concurrentSystems;
2
3 public class LockTest {
4     /**

```

```

5      * Unit test for incrementLocked method
6      * Runs incrementLocked in two threads
7      * @param args
8      */
9
10     public static void main(String args[]) {
11         Incrementer inc = new Incrementer();
12         LockIncRunnable lock1 = new LockIncRunnable(inc, 5);
13         LockIncRunnable lock2 = new LockIncRunnable(inc, 5);
14
15         lock1.start();
16         lock2.start();
17
18         try {
19             lock1.join(15000);
20             lock2.join(15000);
21         } catch (Exception e) {
22             System.out.println("Interrupted");
23         }
24
25         assert inc.getCount() == 10: "Incorrect x value. x = " +
                inc.getCount() + " Expected x = 10";
26     }
27 }

```

## C.4 InterruptTest Class

```

1 package concurrentSystems;
2
3 public class InterruptTest {
4     /**
5      * Unit test for incrementInterrupt method
6      * Runs incrementInterrupt in one thread and incrementLocked in
7      * another
8      * @param args
9      */
10    public static void main(String args[]) {
11        Incrementer inc = new Incrementer();
12        InterruptIncRunnable int1 = new InterruptIncRunnable(inc, 5);
13        LockIncRunnable lock = new LockIncRunnable(inc, 5);

```

```

14         int1.start();
15         lock.start();
16
17         try {
18             int1.join(9000);
19             lock.join(9000);
20         } catch (Exception e) {
21             System.out.println("Interrupted");
22         }
23
24         assert inc.getCount() == 10: "Incorrect count value. Count = " +
25             inc.getCount() + " Expected Count = 10";
26     }
27 }

```

## C.5 AwaitSignalTest Class

```

1 package concurrentSystems;
2
3 public class AwaitSignalTest {
4     /**
5      * Unit test for incrementAwait and incrementSignal methods
6      * Runs incrementAwait in one thread and incrementSignal in another
7      * @param args
8      */
9
10    public static void main(String args[]) {
11        Incrementer inc = new Incrementer();
12        AwaitIncRunnable ai = new AwaitIncRunnable(inc, 3);
13        SignalIncRunnable si = new SignalIncRunnable(inc, 2);
14
15        ai.start();
16        si.start();
17
18        try {
19            ai.join();
20            si.join();
21        } catch (Exception e) {
22            System.out.println("Interrupted");
23        }
24    }

```

```

25         assert inc.getCount() == 5: "Incorrect count value. Count = " +
26             inc.getCount() + " Expected Count = 7";
27     }
}

```

## C.6 SyncTest Class

```

1  package concurrentSystems;
2
3  public class SyncTest {
4      /**
5       * Unit test for SyncIncRunnable run method
6       * Runs synchronized increment methods in two separate threads
7       * @param args
8       */
9
10     public static void main(String args[]) {
11         Incrementer inc = new Incrementer();
12         SyncIncRunnable sync1 = new SyncIncRunnable( inc, 5 );
13         SyncIncRunnable sync2 = new SyncIncRunnable( inc, 5);
14
15         sync1.start();
16         sync2.start();
17
18         try {
19             sync1.join();
20             sync2.join();
21         } catch (Exception e) {
22             System.out.println("Interrupted");
23         }
24
25         assert inc.getCount() == 10: "Incorrect x value. x = " +
26             inc.getCount() + " Expected x = 18";
27     }
}

```