

## Summary

Software is commonly plagued by logic errors, which cause it to behave incorrectly. These errors may not be recognised as such until long after the software is first released. This has prompted substantial work on detecting logic errors. However, less so has been done on correcting them.

This Report documents a Project to develop a software system that both detects and corrects logic errors in models of programs. The corrections can then be applied to the programs themselves. The system covers the following types of logic error:

- Infinite loop
- Infinite recursion
- Arithmetic expression error
- Off-by-one error
- Null dereferencing

The motivation for this Project was originally the Halting Problem encountered in Software engineering, algorithm design and analysis. It suggested that checking whether a loop was infinite by observing its behaviour did not work in the case that it was, because the program would never reach a conclusion. This led to the idea of checking whether a loop was infinite by inferring its behaviour from its initial and final expressions, and its guard.

As technology continues to evolve, it is hoped that software will become more robust against error in general.

Keywords: logic error, computer programming, software tool, static program analysis, dynamic program analysis, model checking, invariant, input validation

# 1 Introduction

## 1.1 Changes from Preliminary Project Report (PPR)

Several aspects of the PPR were changed. The title was modified to describe the Project with greater precision. The experiment performed as part of experimental research shifted focus to infinite loops because unlike most other logic errors addressed by the system, these were corrected by a process of trial-and-error. Deeper insight into null dereferencing led to the decision to address it in the context of string-type instance variables.

## 1.2 Aims

The aims of the Project are:

- To understand how tools are currently being used to address logic errors in computer programming
- To use such tools in new ways

## 1.3 Research questions

The Project sets out to answer the following questions:

- **How can logic errors be detected?** This is about determining whether, given a model of a program, it would behave as expected without needing to run it.
- **Once logic errors have been detected, how can they then be corrected?** This is about adjusting the model upon detecting a logic error. By implementing this adjustment, a program would at least be more likely to behave as expected.

## 1.4 Deliverables

The deliverables from the Software Development Life Cycle (SDLC) are:

- **Survey.** The objective of this deliverable is to gather requirements for a piece of software to be developed as part of the Project.
- **Prototype sketches.** The objective of this deliverable is to outline the design of the software.
- **Software.** The objective of this deliverable is to carry out part of the investigation specified by the Project first-hand.
- **Test suite.** The objective of this deliverable is to ensure that the software is of sufficient quality.
- **Evaluation.** The objective of this deliverable is to identify areas where the Project was successful and where it could be improved.

Among the other deliverables, those that have been submitted are:

- Project proposal form
- PPR
- Initial version of Project Plan
- Literature survey
- Reference list

Those that will be submitted are this Report and a final version of the Project Plan as part of it (Appendix A).

### 1.5 Justification for work done

The work done on the project is important because defects in mission-critical software can spell disaster, like the Ariane 5 incident (Garfinkel 2005). Correct logic is arguably more of a priority than correct syntax because syntax errors are more likely to be addressed than logic errors.

The work is also timely because it addresses a gap in the literature on logic errors (Section 2.3). The results of the survey suggest that the system to be developed would deliver value to programmers (Section 5.1).

Finally, the project lays the foundation for future work (Section 6).

### 3 Methods

#### 3.1 Development methods

##### 3.1.1 Development approaches

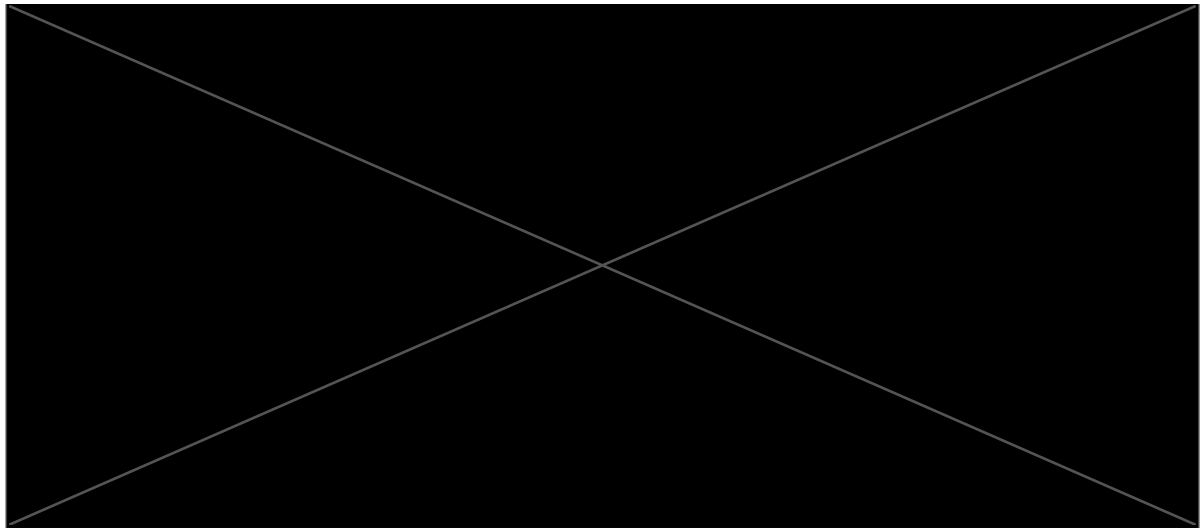
Table 3.1 compares different development approaches.

*Table 3.1: Advantages and disadvantages of different development approaches.*

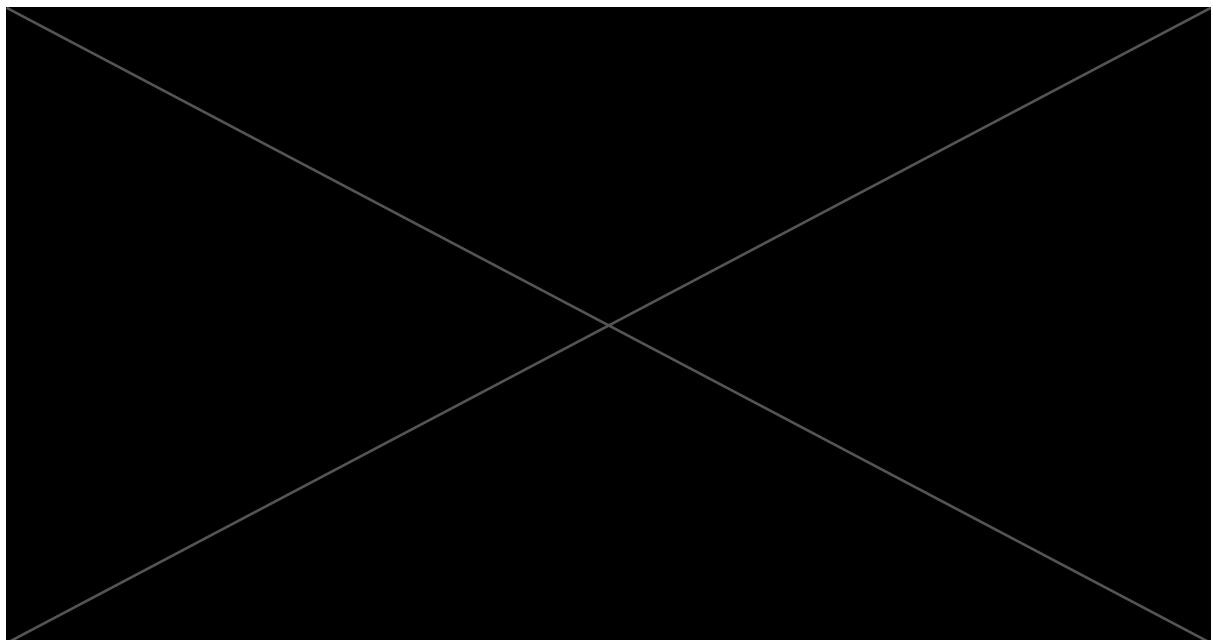
<b>Approach</b>	<b>Advantages</b>	<b>Disadvantages</b>
Waterfall model	<ul style="list-style-type: none"> <li>• Stages in SDLC add structure (Contributor Melonfire 2006)</li> <li>• Complete requirements analysis and design lead to better-quality implementation (ibid)</li> <li>• Facilitates knowledge sharing between differently located staff (ibid)</li> </ul>	<ul style="list-style-type: none"> <li>• Client may not fully understand their requirements at start of project (Contributor Melonfire op. cit.)</li> <li>• Requirements change makes time and cost estimates unreliable (ibid)</li> <li>• Assumes that design that works in theory will work in practice (ibid)</li> </ul>
Evolutionary prototyping	Same as throwaway prototyping, except code can be reused (Dawson 2015, p. 138)	Same as throwaway prototyping, except prototype needs to be designed well from the outset (Dawson op. cit., p. 138)
Throwaway prototyping	<ul style="list-style-type: none"> <li>• Tangibility of prototype makes it convenient to build, use, and discuss (ibid, pp. 137-138)</li> <li>• Facilitates testing of product (p. 137)</li> <li>• Can explore multiple prototypes (ibid)</li> </ul>	<ul style="list-style-type: none"> <li>• Prototype's quality may be low, which may lead final product's quality to be low (ibid, p. 137)</li> <li>• Prototype's target system may not match final product's target system (ibid)</li> <li>• Involves more work than evolutionary prototyping (ibid)</li> </ul>

Evolutionary prototyping was used because 'uncertainty is high' in the project (Hughes & Cotterell 2009 cited in Dawson op. cit., p. 142). Moreover, this is an agile process model (Dawson op. cit., p. 138). This type of process model is suitable for relatively small projects,

such as those undertaken by students individually. If the programming language and deployment environment were unfamiliar, or the requirements well-defined and stable, then throwaway prototyping or the waterfall model could have been used instead (ibid, pp. 141-142). However, this was not the case. Moreover, the project's limited schedule meant that early work could not afford to be discarded and the waterfall's structure discourages changes to the Project Plan, which unforeseen events are likely to cause. Figure 3.1 to Figure 3.3 illustrate the waterfall model, and evolutionary and throwaway prototyping.



*Figure 3.1: Waterfall model (Murray & Sandford 2013, p. 17, Figure 1.2).*



*Figure 3.2: Evolutionary prototyping (Khalid 2018).*

### 3.1.2 Software

The software was programmed in Java because its object-oriented nature encourages good

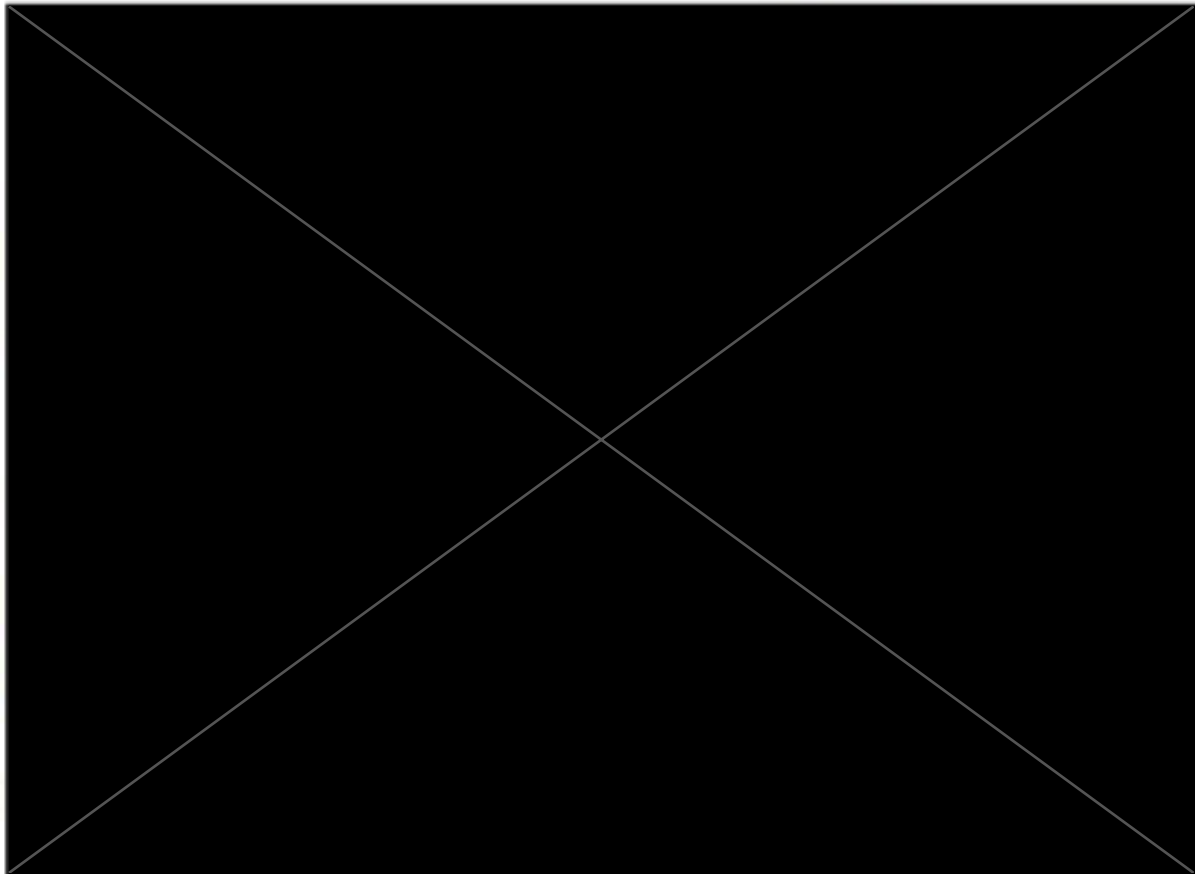


Figure 3.3: Throwaway prototyping (*ibid*).

design practices like reusability, encapsulation, and abstraction. Several solutions in the literature are in the context of this programming language (Section 2.2).

#### 3.1.2.1 Software tools

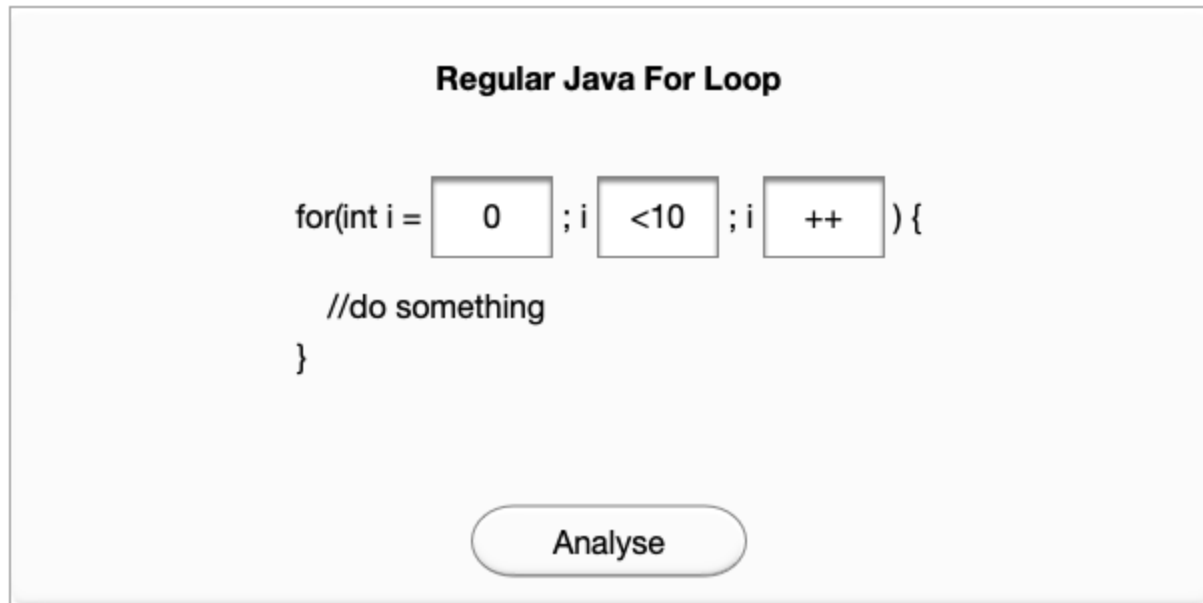
The Google Forms service was used to conduct the survey because it provided automatic data collection and statistical analysis. The NetBeans IDE was used to implement the system, and the Junit framework for testing because both of these tools support Java well. The OmniGraffle design application was used to sketch prototypes because it enabled higher fidelity than hand-drawing.

#### 3.1.2.2 Software libraries

'java.awt' and 'javax.swing' were used to implement a GUI, 'java.math' to model integer sequences, and 'java.util' for various classes. The fundamental library 'java.lang' was used implicitly and 'org.junit' as part of the Junit framework.

#### 3.1.3 Prototype sketches

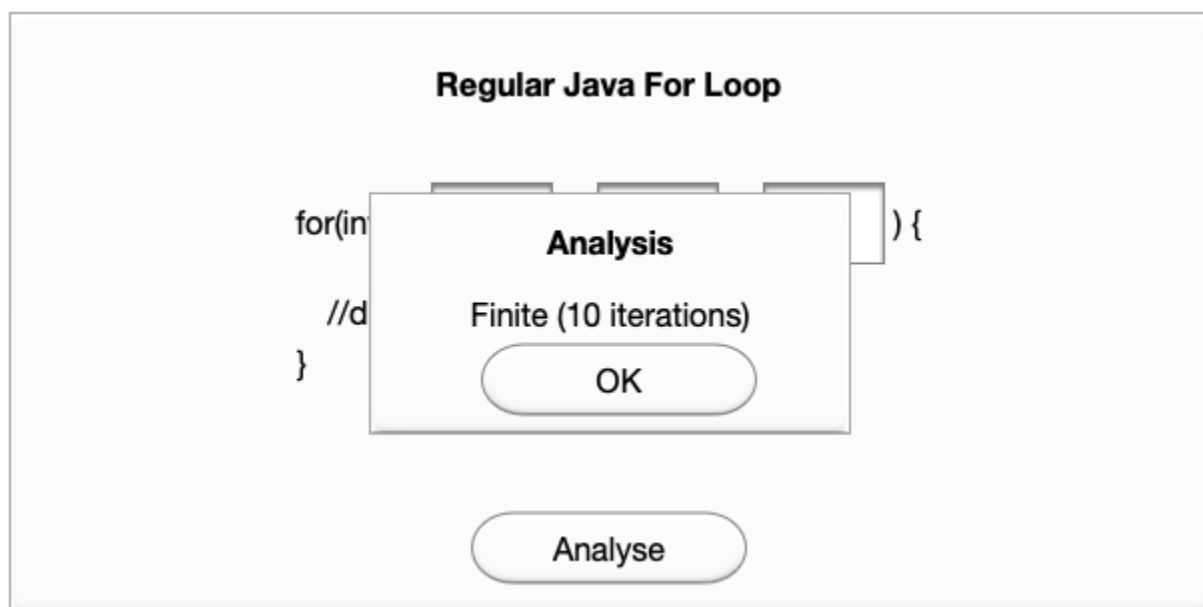
Figure 3.4 to Figure 3.5 show initial design concepts of Java 'for' loop analysis.



**Regular Java For Loop**

```
for(int i =  ; i <  ; i  ) {  
    //do something  
}
```

Figure 3.4: Form where the end user can input a Java 'for' loop.



**Regular Java For Loop**

```
for(int i =  ; i <  ; i  ) {  
    //do something  
}
```

**Analysis**  
Finite (10 iterations)

Figure 3.5: Dialog that shows a termination analysis of the loop and the iteration count.

### 3.1.4 User interaction

Figure 3.6 shows how the end user fundamentally interacts with the system. Appendix B covers this in more detail.

### 3.1.5 Key system functions

Selected code snippets are discussed here. The 'NetBeans\_Project\_Folder\_Documentation.pdf' file in the 'NathanDeFlavis\_120322787\_CO3320\_AdditionalMaterial' directory gives complete code listings.

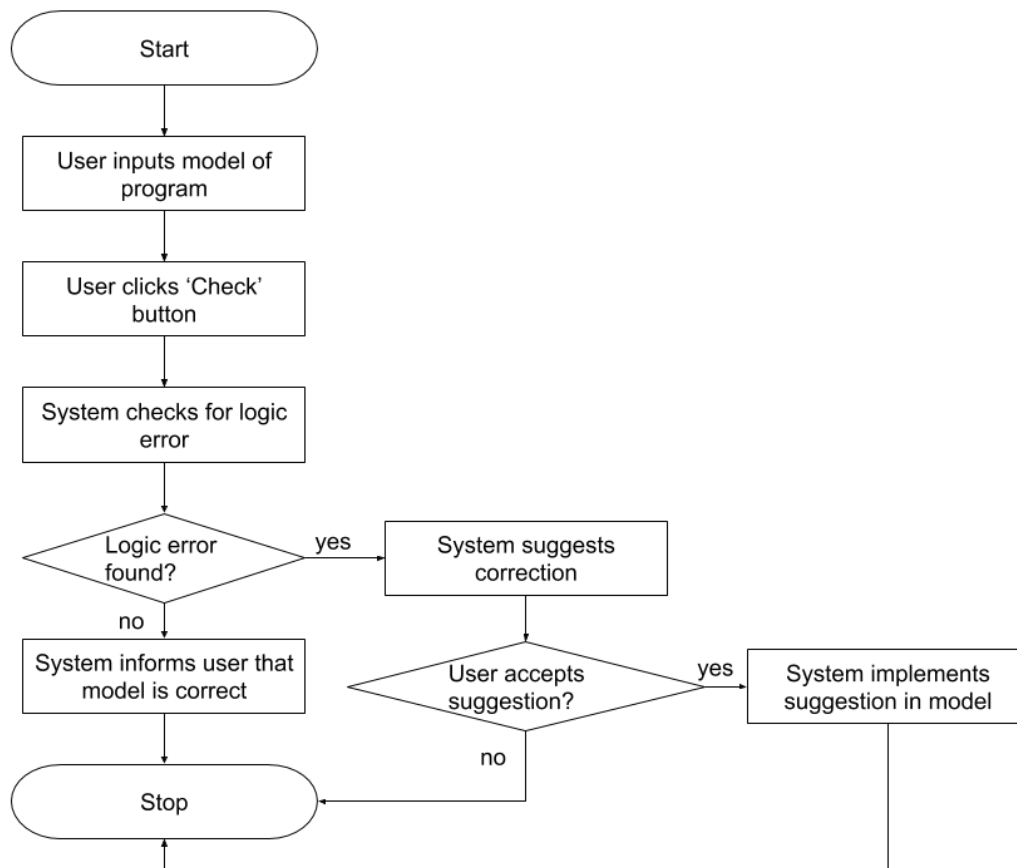


Figure 3.6: Flowchart of user interaction with system.

### 3.1.5.1 Analysing loop termination

The ‘Loop’ class provides a ‘terminates()’ method that can be used for termination analysis.

```

public boolean terminates() {
    return terminatesImmediately() ||
    terminatesEventually();
}

```

The ‘terminatesImmediately()’ method checks whether the loop’s guard would be false at the very start of the loop, in which case the body would not even be run.

```

private boolean terminatesImmediately() {
    int a = is.getA(), b = guard.getB();

    switch (guard.getR()) {

```



```

        case LT:
            return a >= b;
        case LTOET:
            return a > b;
        case GT:
            return a <= b;
        case GTOET:
            return a < b;
        case NET:
            return is.startsWith(b);
        default: //ET
            return !is.startsWith(b);
    }
}

```

If the loop does not terminate immediately, then the ‘terminatesEventually()’ method predicts whether the loop’s guard would become false after a number of iterations.

```

private boolean terminatesEventually() {
    if (is.hasSingleDistinctTerm()) {
        return false;
    } else {
        switch (guard.getR()) {
            case NET:
                return
is.containsNotStartsWith(guard.getB());
            case ET:
                return true;
            default: //LT, LTOET, GT, GTOET
                return
terminatesEventually_neitherNETnorET();
        }
    }
}

```

```
}
```

In certain loops, the guard checks whether the control variable is on one side of the number line. These loops are passed to a special-case method, which itself delegates to abstract methods implemented by subclasses of the ‘Loop’ class.

```
boolean terminatesEventually_neitherNETnorET() {
    switch (guard.getR()) {
        case LT:
        case LTOET:
            return terminatesEventually_LTorLTOET();
        default: //GT, GTOET
            return terminatesEventually_GTorGTOET();
    }
}
```

### 3.1.5.2 *Converting recursive method calls*

The ‘RecursiveMethodCall’ class provides an abstract method, to be implemented by subclasses, which converts a recursive method invocation to an equivalent loop. The ‘getLoopGuard()’ method negates the recursive method’s base case to form the loop’s guard.

```
Guard getLoopGuard() {
    return new Guard((baseCase.getR()).getComplement(),
baseCase.getB());
}
```

### 3.1.5.3 *Generating arithmetic expressions*

The ‘Node’ class models a node in an arithmetic expression tree. It implements Java’s ‘toString()’ method by taking the node as a standalone tree and traversing it to form an arithmetic expression.

```
@Override
public String toString() {
    if (isLeaf()) {
        //term node
```

```

        return label;
    } else {
        //operation node
        Node[] subnodes = new Node[]{left, right};
        int numSubnodes = subnodes.length;
        String[] subnodeStrings = new String[numSubnodes];

        for (int i = 0; i < numSubnodes; i++) {
            Node subnode = subnodes[i];
            String subnodeString = subnode.toString();

            if (!subnode.isLeaf() &&
                (Operation.toOperation(label.charAt(0))).precedes(Operation.to
                Operation((subnode.label).charAt(0)))) {
                subnodeString = "(" + subnodeString + ")";
                //bracket lower-precedence operations
            }

            subnodeStrings[i] = subnodeString;
        }

        return subnodeStrings[0] + " " + label + " " +
        subnodeStrings[1];
    }
}

```

#### 3.1.5.4 *Generating loops to traverse array index ranges*

The ‘ArrayPositionRange’ class models a contiguous range of 1-based positions in an array. It provides a ‘getTraversingLoops()’ method to generate loops to traverse the corresponding 0-based array index range.

```

public List<Loop> getTraversingLoops() {
    int a = start - 1, b = end - 1, d;
    List<Loop> loops = new ArrayList<>();
    List<Guard> guards = new ArrayList<>();
}

```

```

    if (a > b) {
        guards.add(new Guard(Relation.GT, b - 1));
        guards.add(new Guard(Relation.GTOET, b));
        d = -1;
    } else {
        guards.add(new Guard(Relation.LT, end));
        guards.add(new Guard(Relation.LTOET, b));
        d = 1;
    }

    guards.stream().forEach((guard) -> {
        loops.add(new ArithmeticLoop(new
ArithmeticInfiniteSequence(a, d), guard));
    });

    return loops;
}

```

### 3.1.5.5 Explicitly initialising string-type instance variables

The ‘Metaobject’ class models an object-oriented class that declares one or more string-type instance variable(s). It provides a ‘getUninitialisedVars()’ method to check for cases where a constructor omits to explicitly initialise a variable.

```

public List<String> getUninitialisedVars() {
    List<String> uninitialisedVars = new ArrayList<>();

    for (String var : vars) {
        for (Map<String, String> initialState :
initialStates) {
            if (!initialState.containsKey(var)) {
                uninitialisedVars.add(var);
                break;
            }
        }
    }
}

```

```

        }
    }

    return uninitialisedVars;
}

```

If any cases are found, the ‘addMissingInitialisations()’ method can be used to explicitly initialise a variable wherever a constructor omits to.

```

public void addMissingInitialisations() {

    getUninitialisedVars().stream().forEach((uninitialisedVar) ->
    {
        initialStates.stream().filter((initialState) ->
        (!initialState.containsKey(uninitialisedVar))).forEach((initialState) -> {
            initialState.put(uninitialisedVar, "\\\"");
        });
    });

    List<Map<String, String>> distinctInitialStates = new
    ArrayList<>();

    //remove duplicate constructors
    initialStates.stream().filter((initialState) ->
    (!distinctInitialStates.contains(initialState))).forEach((initialState) -> {
        distinctInitialStates.add(initialState);
    });

    initialStates = distinctInitialStates;
}

```

## 3.2 Research methods

Table 3.2 compares different research methods.

Table 3.2: Advantage and disadvantage of different research methods.

Method	Advantage	Disadvantage
Experimental research	Tests are controlled by experimenter (Dawson 2015, p. 28)	May be limited by sampling and ethical issues (ibid)
Action research	Addresses an existing problem (Herbert 1990 cited in Dawson loc. cit.)	May focus on action rather than its evaluation (Dawson loc. cit.)
Case study research	Can be performed both directly and indirectly (Dawson loc. cit.)	No need to process large amount of data in order to make sense of it (ibid, p. 29)

Experimental research methods were used because as a development-based project, the implemented system can be used to test out a hypothesis in practice (Dawson op. cit., p. 118). If the project were associated with a specific organisation, then case study or action research methods could be used (ibid, p. 28). However, this is not the case. Moreover, these methods lack the technical focus required by the project. Figure 3.7 to Figure 3.9 illustrate action, case study, and experimental research.

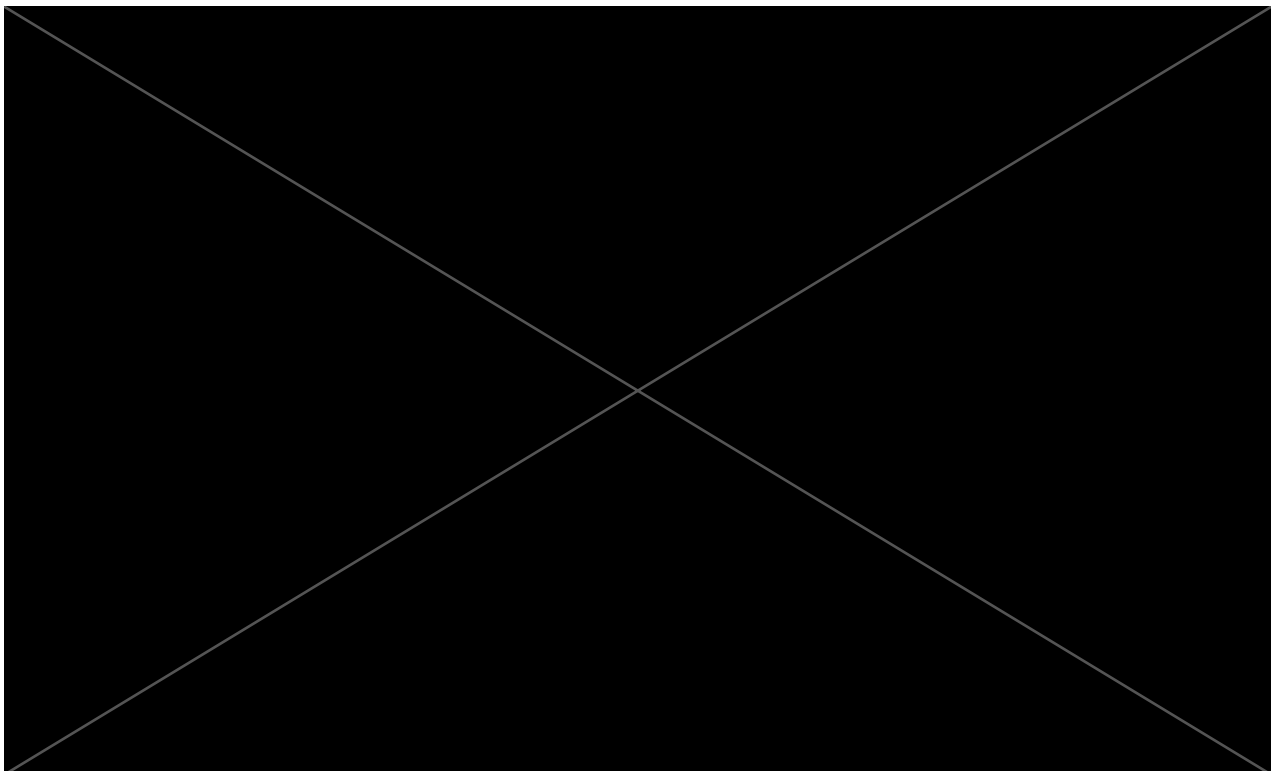
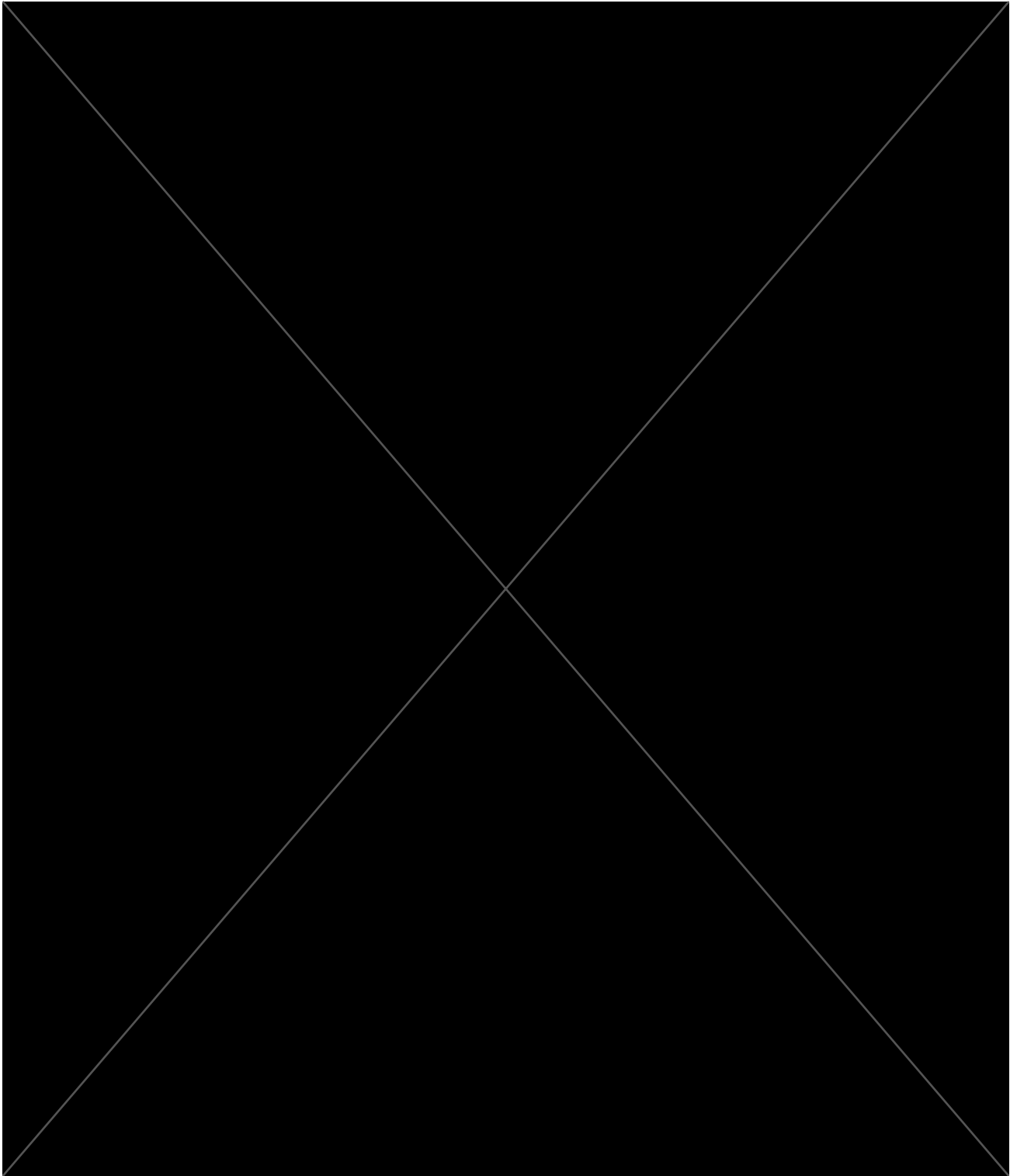


Figure 3.7: Action research (Riel 2019, Figure 1).

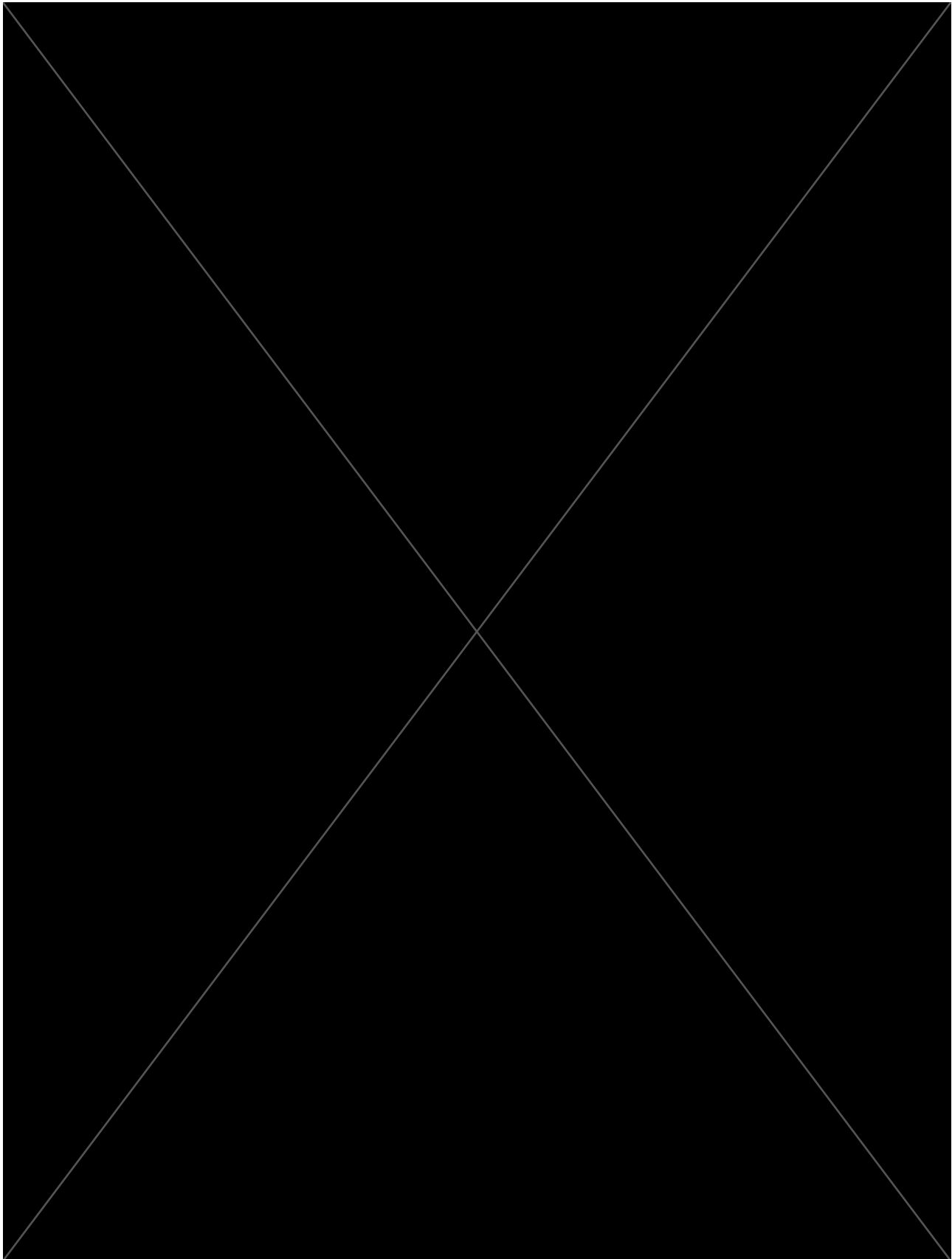
The experiment performed as part of experimental research was adapted from a coin-flipping example (Wikipedia 2019). For the purpose of this experiment:

- An arithmetic loop is a loop for which the values that the control variable assumes form an arithmetic sequence.



*Figure 3.8: Case study research (Ramli 2015).*

- A proper correction is a loop for which the body iterates at least twice.
- A is the event that given an infinite arithmetic loop to correct, the system generates at least 1 proper correction.



*Figure 3.9: Experimental research (EdrawSoft n. d.).*

The null hypothesis  $H_0$  is that whether A occurs is random, in which case  $P(A) = 0.5$ . The test



statistic  $X$  is the frequency with which  $A$  occurs, and the level of significance  $\alpha$  is 0.05.

Survey research methods were used to gather requirements. A questionnaire was conducted because the system is targeted at a user group rather than a single user. The format was as follows:

## QUESTIONNAIRE TO COMPUTING STUDENTS

Dear respondents, I am **Nathan De Flavis**, a student of Computing and Information Systems at University of London. I am carrying out research entitled '**An investigation into logic errors in computer programming: how tools can be used**'.

I therefore kindly request for you to fill in this questionnaire to assist me in the completion of my research. I guarantee that all information given shall be treated with confidentiality and used for academic purposes only.

**Instruction:** Please tick where applicable.

1. In the context of Java programming, which bug do you encounter most often?
    - ☐ Infinite loop
    - ☐ StackOverflowError
    - ☐ When iterating through an array, not stopping at the last element
    - ☐ In a multithreaded program, uncoordinated threads
    - ☐ NullPointerException
    - ☐ Arithmetic error
  2. When do you usually detect bugs?
    - ☐ As I type
    - ☐ At compile time
    - ☐ At runtime
  3. When you detect bugs, do you know how to fix them?
    - ☐ Yes
    - ☐ No
  4. Do you spend a significant amount of time debugging programs?
    - ☐ Yes
    - ☐ No
  5. When do you prefer to have code checked for syntax errors?
    - ☐ As I type
    - ☐ At compile time
- ☐ I hereby give my permission for the student to quote my responses in a CO3320 Project report with the understanding that:
- This report will be submitted to the university of London.
  - I waive any claim to copyright of this material should the student ever publish it in an electronic format online.
  - The student will maintain my anonymity as part of this questionnaire.

The research process was circulatory overall but evolutionary in terms of investigation because this view most accurately describes research in practice (Dawson 2015, p. 23).

## 4 Results

### 4.1 Requirements gathering

Figure 4.1 to Figure 4.5 show the results of the survey.

**In the context of Java programming, which bug do you encounter most often?**

5 responses

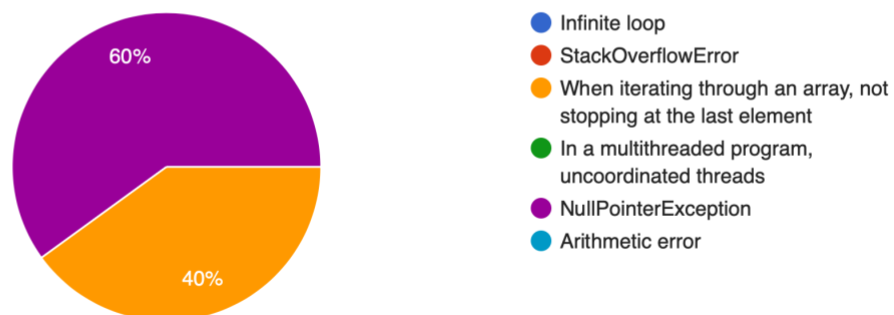


Figure 4.1: Question 1 responses.

**When do you usually detect bugs?**

5 responses

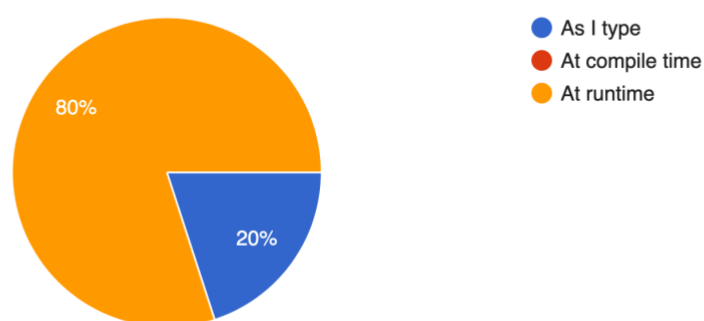


Figure 4.2: Question 2 responses.

### 4.2 Testing

Figure 4.6 shows the results of the system's unit tests. Figure 4.7 shows a set of acceptance

### When you detect bugs, do you know how to fix them?

5 responses

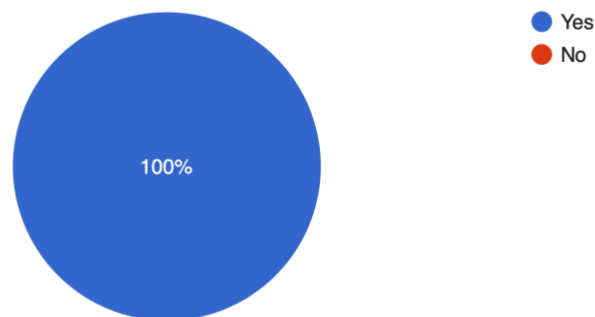


Figure 4.3: Question 3 responses.

### Do you spend a significant amount of time debugging programs?

5 responses

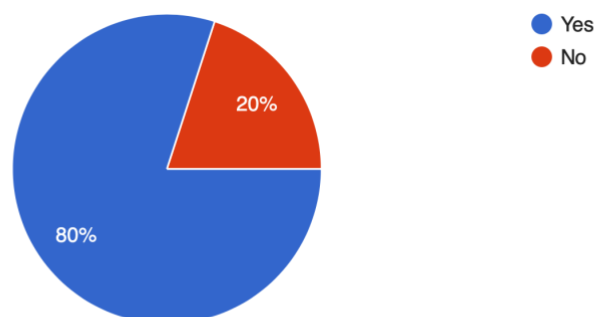


Figure 4.4: Question 4 responses.

test cases.

## 4.3 Experiment

The observation  $O$  is 26 occurrences of  $A$  out of 100 infinite loop spawns. The right-tailed p-value of  $O$ , given that  $H_0$  is true, is:

$$\begin{aligned}
 P(X \geq 26) &= P(X = 26) + P(X = 27) + \dots + P(X = 100) \\
 &= \frac{\binom{100}{26} + \binom{100}{27} + \dots + \binom{100}{100}}{2^{100}} \\
 &= 0.9999999718
 \end{aligned}$$

## When do you prefer to have code checked for syntax errors?

5 responses



Figure 4.5: Question 5 responses.

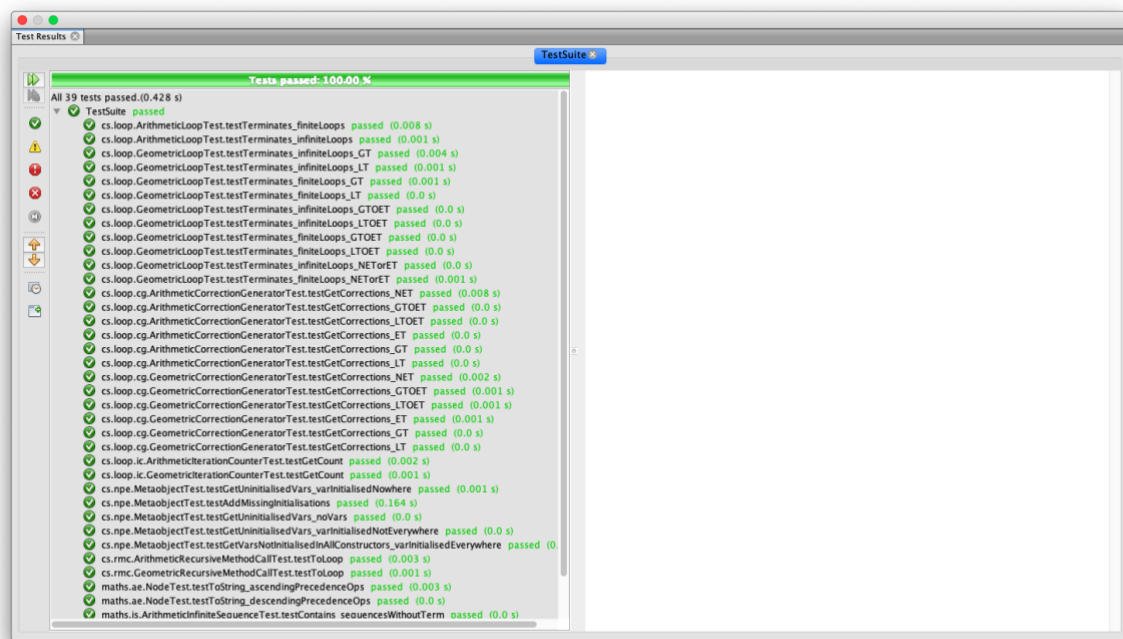


Figure 4.6: Unit test results.

Requirement to test	Input(s)	Expected output	Actual output
Check for infinite loop error	a = 10, R = '>', b = 0, operation = '+', d = -1	No error	No error
	a = 10, R = '>', b = 0, operation = '+', d = 1	Error	Error
Check for infinite recursion	a = 10, R = '<', b = 0, operation = '+', d = -1	No error	No error
	a = 10, R = '<', b = 0, operation = '+', d = 1	Error	Error
Check for arithmetic expression error	Instructions = {ADD(a, b), STORE, ADD(c, d), DIVIDE(STORAGE, RESULT)}, Expression = '(a + b) / (c + d)'	No error	No error
	Instructions = {ADD(a, b), STORE, ADD(c, d), DIVIDE(STORAGE, RESULT)}, Expression = 'a + b / c + d'	Error	Error
Check for off-by-one error	Start position = 1, End position = 10, a = 0, R = '<', b = 10, d = 1	No error	No error
	Start position = 1, End position = 10, a = 1, R = '<', b = 10, d = 1	Error	Error
Check for null pointer error	Variables = {s1, s2}, Constructors = {c1, c2}, Initialisations = {(c1, s1, "something"), (c1, s2, "")}, (c2, s1, ""), (c2, s2, "something")	No error	No error
	Variables = {s1, s2}, Constructors = {c1, c2}, Initialisations = {(c1, s1, "something"), (c2, s2, "something")}	Error	Error

Figure 4.7: Acceptance test results.

## 5 Discussion

### 5.1 Requirements gathering

The results were collected in a spreadsheet linked to the questionnaire as an online form. When the respondent submitted the form, their response was automatically recorded in the spreadsheet.

Figure 4.1 shows that the most common error in Java encountered by respondents was null dereferencing. In the context of all object types, this was found to be too complex to address in the time available. Therefore, the project addressed null string dereferencing only.

Figure 4.2 shows that most respondents did not detect logic errors until runtime. This suggests that static program analysis would be beneficial over dynamic analysis in catching mistakes earlier. Therefore, the project used static analysis mainly and dynamic analysis to a lesser extent.

Figure 4.3 shows that the respondents know how to correct logic errors. This suggests that they would not benefit as much from error correction on the part of the tool. However, in a review of the Project, the Head of Programmes noted that the tool could help novice programmers recover from mistakes. Therefore, logic error correction was implemented as part of the Project.

Figure 4.4 shows that most respondents spend a significant amount of time debugging programs. This suggests that expediting this process would improve productivity. Therefore, the system would deliver value to programmers.

Figure 4.5 shows that the respondents prefer to check for errors during typing. This suggests that the system would be useful as a plugin to an IDE. However, the Head of Programmes felt that only a standalone tool would be viable given the Project's time constraint. Therefore, the system was developed as a standalone tool, and a plugin version left as future work (Section 6).

### 5.2 Testing

It is worth noting that testing software can only prove the presence of some bugs, not the absence of all bugs.

#### 5.2.1 Unit testing

For most units in the system, Junit tests were written and run as a test suite. The IDE then displayed the results of running the tests.

Figure 4.6 shows that all unit tests passed. This suggests that the units that were tested work correctly. Some units were excluded from testing because they were either untestable or too trivial.

#### 5.2.2 Acceptance testing

For each of the logic errors addressed by the system, a positive test case and a negative test case were specified. The test cases were performed by interacting with the program via the UI. The system then displayed the output of performing the test cases.

Figure 4.7 shows that all acceptance tests passed. This suggests that the system has been validated to meet the requirements.

### 5.3 Experiment

A class was written to randomly generate arithmetic loops and keep track of those that were infinite. For each infinite loop, corrections were generated and if any was a proper correction, an observation was counted. At the end, the number of observations was displayed and used to compute the p-value.

The p-value exceeds  $\alpha$ , so  $H_0$  is accepted. This suggests that given an infinite arithmetic loop to correct, it is random as to whether the system generates at least 1 proper correction.

It is possible that not all the loops were unique because this was not checked. However, the chance of 2 of these loops being identical is only:

$$\frac{1}{[(2^{32})^3 * 6]^2}$$

## 6 Conclusion

The Project has demonstrated that software tools can be used to both detect and correct the following kinds of logic error:

- Infinite loops of the form:

```
i := a
```

```
WHILE i R b
    //do something with i
    i := i + d
END_WHILE
```

- Infinite loops of the form:

```
i := a
```

```
WHILE i R b
    //do something with i
    i := i * r
END_WHILE
```

- Infinite recursion of the form:

```
METHOD recursiveMethod(i)
    IF I R b THEN
        //don't recurse
    ELSE
        //call recursiveMethod(i + d)
    END_IF
END_METHOD
```

```
//at some point in program, call recursiveMethod(a)
```

- Infinite recursion of the form:

```
METHOD recursiveMethod(i)
    IF I R b THEN
        //don't recurse
    ELSE
        //call recursiveMethod(i * r)
    END_IF
END_METHOD

//at some point in program, call recursiveMethod(a)
```

- Mismatch between an arithmetic expression and the result of an arithmetic computation
- Mismatch between the control variable range of a loop that accesses an array at 0-based indices within this range, and the range of 1-based positions at which the loop is intended to access the array
- A constructor's omission to initialise a string-type instance variable that is declared without a value and subsequently dereferenced

The objectives of the Project were met for the most part (Evaluation Section). Future work includes:

- Building a plugin version of the tool
- Detecting logic errors in Java code (later as the user types)
- Addressing null dereferencing for all object types (later other kinds of logic error)



## Appendix A Final Project Plan

Figure A.1 shows how the project work was actually scheduled, which was significantly different from the original project plan. The task of getting sample source code was removed because it was never used. The literature review is part of both the literature survey and completing the FPR.

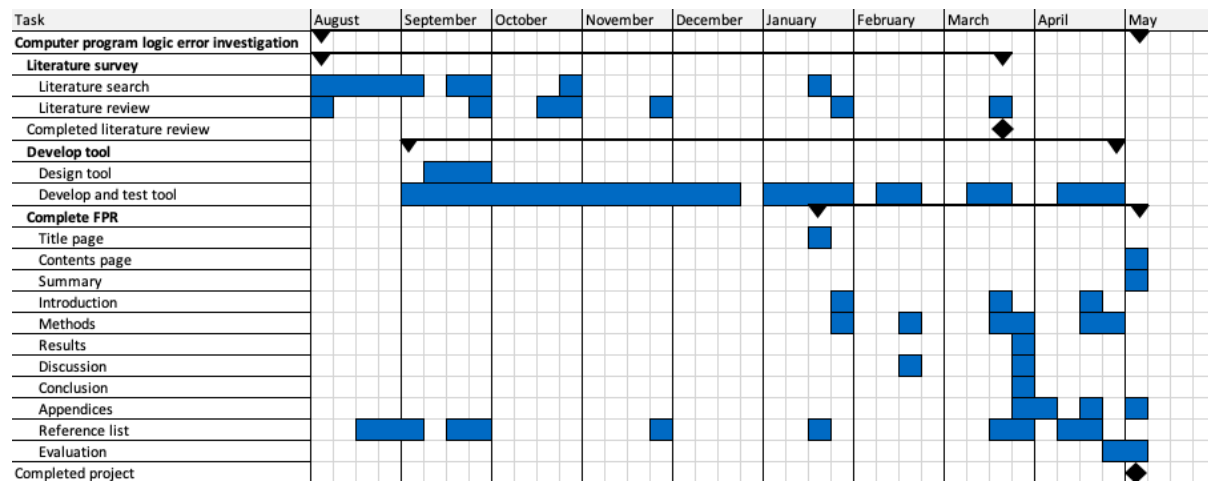


Figure A.1: Final version of Project Plan.

## Appendix B User guide

The structure of this guide was adapted from Dawson (2015, pp. 233-234).

### B.1 Overview of software

This software detects and corrects logic errors in models of object-oriented programs. It is intended especially for Java developers but may benefit developers programming in other languages.

### B.2 Minimum hardware requirements

Storage requirements may vary but at least 128MB of available RAM and 2MB of available hard drive capacity are recommended. A Java-compatible desktop operating system is required.

### B.3 Installing software

Only the Java Runtime Environment (JRE) (version 8 or later) needs to be installed to run the software. The JRE is downloadable from

<https://www.java.com/inc/BrowserRedirect1.jsp?locale=en>.

### B.4 Starting software

To start the software:

- Navigate to the directory 'NathanDeFlavis\_120322787\_CO3320\_AdditionalMaterial' > 'LogicErrorTool' > 'dist' and double-click the 'LogicErrorTool.jar' file.
- Alternatively, the software can be run at the command line. The 'README.txt' file, in the same directory, explains how to do this.

### B.5 Using software

#### B.5.1 Infinite loop/recursion error

1. Enter integer values for 'a', 'b', and 'd' / 'r'.
2. Select a relation operator for 'R'.
3. Select an arithmetic operator between 'i' and 'd' / 'r'.
4. Click 'Check'.

If the loop/recursion is finite:

5. Click 'OK'.

If the loop/recursion is infinite:

5. Select a correction.

6. Click 'Change'.

Figure B.1 and Figure B.2 demonstrate how a user would use the 'Infinite loop' screen.

The screenshot shows a window titled "LogicErrorTool 2.0". At the top, there are five tabs: "Infinite loop", "Infinite recursion", "Arithmetic expression", "Off-by-one", and "Null pointer". The "Infinite loop" tab is selected. Below the tabs, the configuration is as follows:

- Variable **a** is assigned the value **10** (`i := 10`).
- A **WHILE** loop with condition **R** (`>`) and value **b** (`0`).
- Inside the loop, there is a comment `//do something with i`.
- After the loop, variable **d** is assigned the value `i + 1` (`i := i + 1`).
- The loop ends with `END_WHILE`.
- A **Check** button is at the bottom.

Figure B.1: The user inputs a loop and clicks 'Check'.

The screenshot shows a window titled "LogicErrorTool 2.0" with the text "Loop is infinite" and "Suggestions". Below this is a table with five columns: **a**, **R**, **b**, **d**, and **Iterations**.

a	R	b	d	Iterations
-10	>	0	1	0
10	<	0	1	0
10	>	0	-1	10

At the bottom of the window is a **Change** button.

Figure B.2: The system finds an infinite loop error and suggests corrections. The user selects one and clicks 'Change'.

### B.5.2 Arithmetic expression error

1. Add instructions as follows:
  - a. Select an operation under 'Operation'. NB: The 'RESULT' variable refers to the result of the last instruction. The 'STORE' operation temporarily stores a copy of 'RESULT' in the 'STORAGE' variable.
  - b. Enter expressions for 'a' and 'b'.
  - c. Click 'Add'.
2. Enter an arithmetic expression that the instructions are expected to generate.
3. Click 'Check'.

If the arithmetic expression is correct:

4. Click 'OK'.

If the arithmetic expression is incorrect:

4. Click 'Yes'.

Figure B.3 and Figure B.4 demonstrate how a user would use the 'Arithmetic expression' screen.

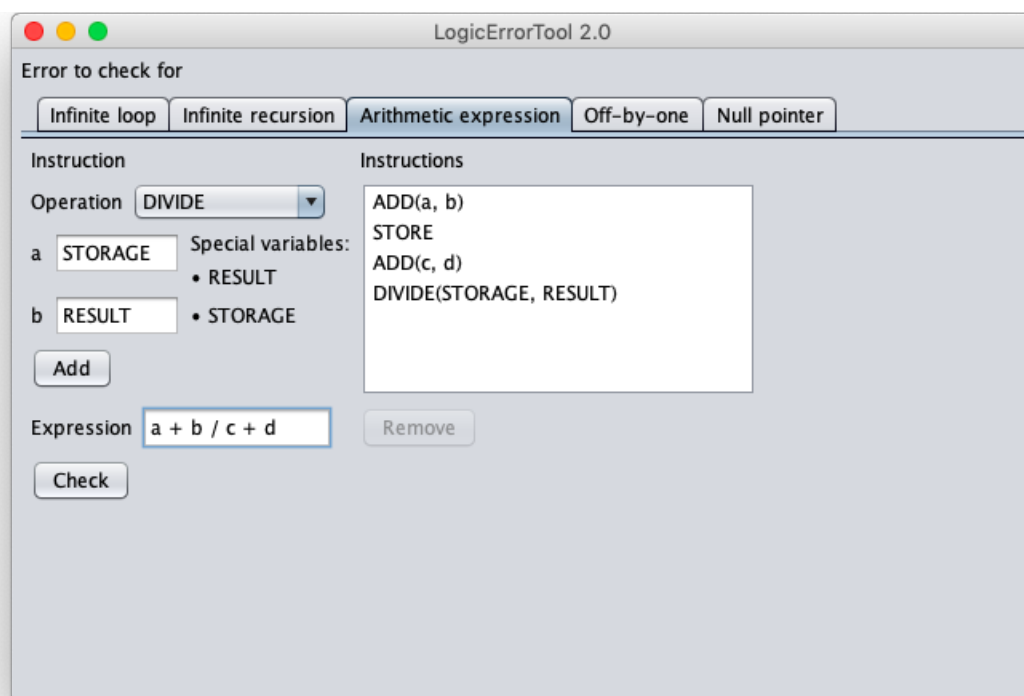


Figure B.3: The user inputs a set of instructions and the arithmetic expression they expect the instructions to generate. They then click 'Check'.

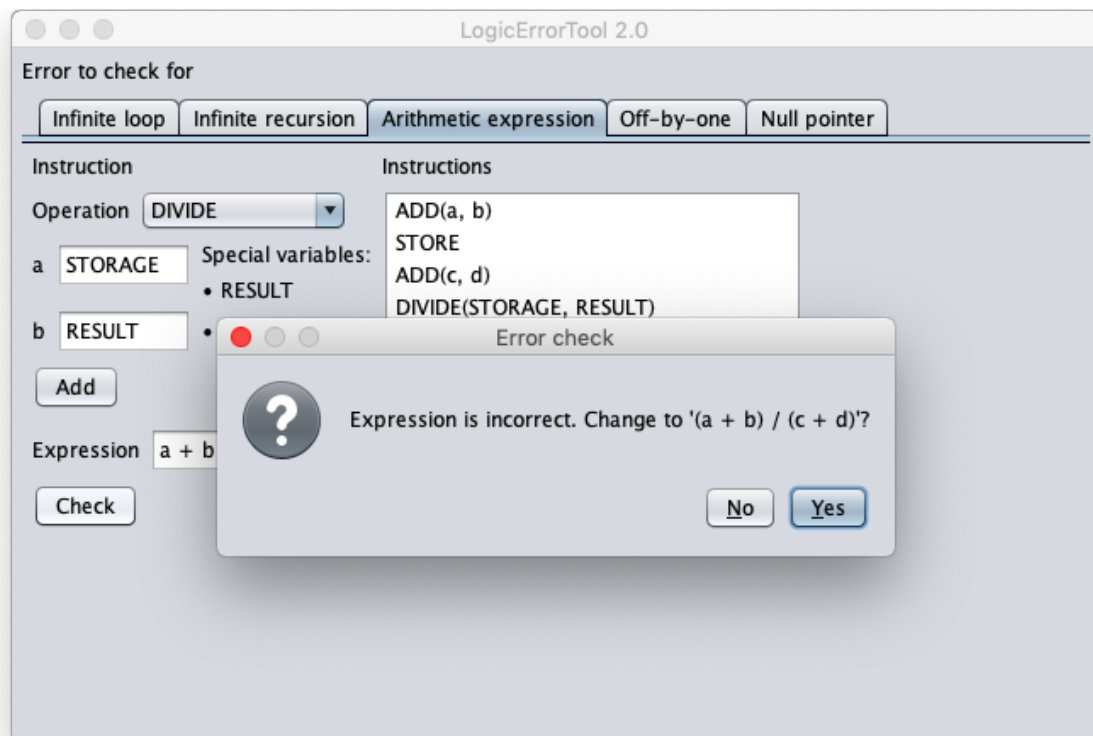


Figure B.4: The system finds an arithmetic expression error and suggests a correction. The user clicks 'Yes'.

### B.5.3 Off-by-one error

1. Enter values for 'Start position', 'End position', 'a', and 'b'.
2. Select a relational operator for 'R'.
3. Select a value for 'd'.
4. Click 'Check'.

If the loop is correct:

5. Click 'OK'.

If the loop is incorrect:

5. Select a correction.
6. Click 'Change'.

Figure B.5 and Figure B.6 demonstrate how a user would use the 'Off-by-one' screen.

### B.5.4 Null pointer error

1. Add variables as follows:
  - a. Enter a value for 'Name'.

LogicErrorTool 2.0

Error to check for

---

Array position range (first position is 1)

Start position

End position

Expected loop to traverse corresponding array index range

a

Initialisation: i =

R      b

Guard: i

d

Final expression: i = i +

Figure B.5: The user inputs an array position range and a loop they expect to traverse the corresponding array index range. They then click 'Check'.

Loop is incorrect (likely off-by-one error)

Suggestions

a	R	b	d
0	<	10	1
0	≤	9	1

Figure B.6: LET finds an error that is likely off-by-one and suggests corrections. The user selects one and clicks 'Change'.

- b. Click 'Add' under 'Name'.
2. To add constructors other than the default constructor, click 'Add' under 'Constructors'.
3. Add initialisations of a variable in a constructor as follows:

- a. Select a variable.
  - b. Select a constructor.
  - c. Enter a value for 'Selected variable: Value'.
  - d. Click 'Add' below 'Selected variable: Value'.
4. Click 'Check'.

If there is no null pointer error:

5. Click 'OK'.

If there is a null pointer error:

5. Click 'Yes'.

Figure B.7 and Figure B.8 demonstrate how a user would use the 'Null pointer' screen.

LogicErrorTool 2.0

Error to check for

Infinite loop Infinite recursion Arithmetic expression Off-by-one **Null pointer**

Object-oriented class

String instance variables

s1
s2

Constructors

c1
c2

Selected constructor: Initialisations

Name	Value
s2	"something"

Remove Add Remove Remove

Name  Selected variable: Value:

Add Add

//at some point in program, instantiate class and call string instance method on each variable

Check

Figure B.7: The user inputs a set of string instance variables, constructors, and initialisations to perform. They then click 'Check'.

## B.6 Ending and uninstalling software

To stop running the software, close the 'LogicErrorTool' window.

Only the JRE can be uninstalled. To uninstall the JRE, see <https://www.java.com/en/uninstall/>.

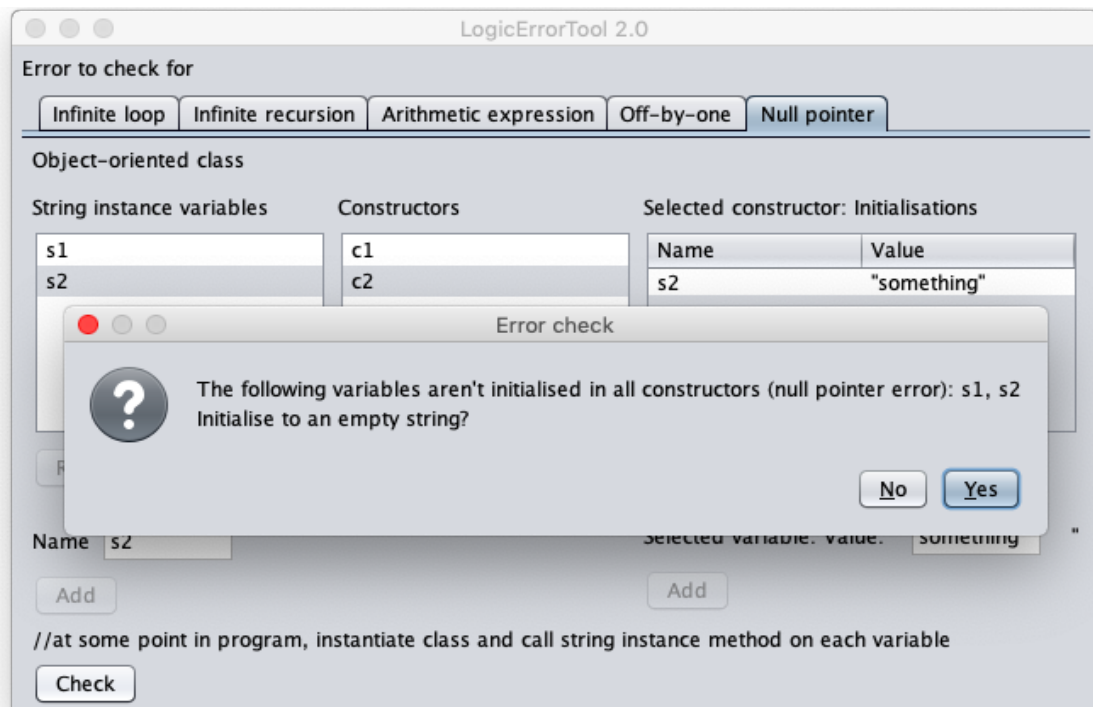


Figure B.8: LET finds a null pointer error and suggests a correction. The user clicks 'Yes'.

## B.7 Known issues

Corrections and iteration counting may be inaccurate for loops with inputs of large magnitude. Any other issues encountered can be reported to [ngdf1@student.london.ac.uk](mailto:ngdf1@student.london.ac.uk).



## Reference list

- Brumley, D, Chiueh, T, Johnson, R, Huijia, L, Song, D 2007, *RICH: Automatically Protecting Against Integer Vulnerabilities*, viewed 20 September 2018, <https://web.archive.org/web/20121010025025/http://www.cs.cmu.edu/~dbrumley/pubs/integer-ndss-07.pdf>
- Contributor Melonfire 2006, *Understanding the pros and cons of the Waterfall Model of software development*, viewed 17 April 2019, <https://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/>
- Cornu, B, Barr, E, Seinturier, L, Monperrus, M 2016, *Casper: Automatic Tracking of Null Dereferences to Inception with Causality Traces*, viewed 18 January 2019, <https://hal.archives-ouvertes.fr/hal-01354090/document>
- Dawson, C 2015, *Projects in Computing and Information Systems: A Student's Guide*, 3rd edn, Pearson Education Ltd, Harlow
- EdrawSoft n. d., *Do Experiment Flowchart Template*, viewed 22 March 2019, <https://www.edrawsoft.com/template-do-experiment-flowchart.php>
- Epp, S 2011, *Discrete Mathematics with Applications*, 4th edn, Brooks/Cole Cengage Learning, Boston
- Felmetsger, V, Cavedon, L, Kruegel, C, Vigna, G 2010, *Toward Automated Detection of Logic Vulnerabilities in Web Applications*, viewed 23 August 2018, [https://pdfs.semanticscholar.org/0a9f/ebbf3daff2db95accc73eb74d5dd05b54cb8.pdf?\\_ga=2.65005487.1822630563.1556777905-1404149658.1556777905](https://pdfs.semanticscholar.org/0a9f/ebbf3daff2db95accc73eb74d5dd05b54cb8.pdf?_ga=2.65005487.1822630563.1556777905-1404149658.1556777905)
- Garfinkel, S 2005, *History's Worst Software Bugs*, viewed 29 January 2019, <https://www.wired.com/2005/11/historys-worst-software-bugs/?currentPage=all>
- Khalid, H 2018, *Difference Between Evolutionary Prototyping and Throw-away Prototyping*, viewed 22 March 2019, <https://prototypeinfo.com/evolutionary-prototyping-and-throw-away-prototyping/>
- Murray, D, Sandford, N 2013, *Software engineering project management*, University of London, London
- Pellegrino, G, Balzarotti, D 2014, *Toward Black-Box Detection of Logic Flaws in Web Applications*, viewed 23 August 2018, [https://pdfs.semanticscholar.org/b38a/e64e42cab17662e2d04b90c09cb8c82f2677.pdf?\\_ga=2.74528404.767654814.1535013533-473936964.1534251100](https://pdfs.semanticscholar.org/b38a/e64e42cab17662e2d04b90c09cb8c82f2677.pdf?_ga=2.74528404.767654814.1535013533-473936964.1534251100)
- Ramli, N 2015, *Single case study research - Easy flowchart*, viewed 22 March 2019, <https://www.slideshare.net/suhailiramli/single-case-study-research-easy-flowchart>
- Riel, M 2019, *Understanding Collaborative Action Research*, viewed 22 March 2019, <http://cadres.pepperdine.edu/ccar/define.html>
- Silberschatz, A, Galvin, P, Gagne, G 2012, *Operating System Concepts*, 9th edn, John Wiley & Sons Inc., Hoboken

- Stergiopoulos, G, Katsaros, P, Gritzalis, D 2014, *Automated detection of logical errors in programs*, viewed 15 August 2018,  
<https://www.infosec.aueb.gr/Publications/CRISIS-2014%20Logical%20Errors.pdf>
- Turing, A 1936, *On computable numbers, with an application to the Entscheidungsproblem*, viewed 6 September 2018,  
[https://www.cs.virginia.edu/~robins/Turing\\_Paper\\_1936.pdf](https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf)
- Wheeler, D 2004, *Secure programmer: Prevent race conditions*, viewed 28 September 2018,  
<https://www.ida.liu.se/~TDDC90/literature/papers/SP-race-conditions.pdf>
- Wikipedia 2019, *p-value*, viewed 15 March 2019,  
[https://en.wikipedia.org/w/index.php?title=P-value#Coin\\_flipping](https://en.wikipedia.org/w/index.php?title=P-value#Coin_flipping)

## Evaluation

This work has been effective for simple program models but may not scale well for complex source code. It is most useful for novice programmers, who when encountering a logic error may not know how to recover from it. These users would benefit from the guidance provided by the system.

The objectives were achieved to a fair extent. The survey's response rate was lower than expected but the responses provided insight into what the scope of the Project should be. The prototype sketches were not as elaborate as the final system, but they formed the basis for the infinite loop part of the UI. The system addressed certain instances of infinite loops and recursion, and null dereferencing, but scaled better for arithmetic expression errors. Code coverage within methods was not measured but the test suite covered all of the nontrivial methods in the system.

I am pleased with the quality of the work, but I see ways it could be enhanced. In hindsight, I realise that I was too eager to move to the development part and on future projects, I would spend more time planning and documenting work done. I hope to have the opportunity to develop this system for a specific client. This would make requirements gathering and acceptance testing more reliable.

Overall, the Project has been a valuable learning experience in developing research skills, managing time more effectively, and thinking laterally to solve problems.