How to provide automated feedback helping students with negative semantic transfer when learning a second programming language

Jenny Strømmen

Supervisors: Anya Helene Skrove Bagge and Anna Maria Eilertsen

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2022

Abstract

Earlier studies have shown that when students see matching syntax across programming languages, they believe the semantics will match. Typically this is true, but occasionally the syntax between two languages is similar while the semantics are different. Given that the syntax in Java is correct, the code will compile with no warnings, and the cause of the error can take a longer time to find and be harder to correct. This thesis collects six semantic errors in Java with no preexisting error message that might be problematic for a student when transferring from Python to Java.

We aim to find out if the errors are a problem for the students and uncover that current environments lack feedback we believe is beneficial for novice Java students. We develop a tool, Uncoil, to detect the errors and provide an error message to fill this gap. Seven novice students in Java with previous Python knowledge tried to solve the errors and evaluated Uncoil in a mixed method study. Our results indicate that novice Java students need help with some of the errors earlier in the semester but do not transfer the semantics from Python to Java later in the semester. At the time of the study, few students needed Uncoil to solve the errors, but especially the weaker students found it helpful.

"*From then on, when anything went wrong with a computer, we said it had bugs in it.*"

— *Grace Hopper*

Acknowledgements

First and foremost, I would like to thank my supervisors, Anya Helene Bagge and Anna Eilertsen, for their support and guidance.

My interest in the subject came from Anya when she proposed to create a bug tool for Java. Having experienced the transition from Python to Java, I knew I wanted to help students with the transfer. Reading previous research I found that semantics errors took the longest time to solve for the students, and that programmers transfer the semantics between programming languages. I wanted to find out if students that knew Python before could transfer the semantics onto Python-like syntax in Java. Finding research to support this, I wanted to help the students with this transfer and find out what errors were at the highest risk for the students to make.

I want to thank my partner Espen for his uplifting conversations and support throughout this time. Thank you, parents, Merethe and Kaj, stepparents Peter and Brit and my brothers Martin and Hans Kristian, for taking an interest in the subject and motivating me. I want to thank my friend Tellev for helping to recruit students for the study and discussions on the subject. I want to thank my former coworker Sindre for helpful feedback. Thank you, Magne Haveraaen, for your tips about the topic. I also want to thank the participants in the focus group. I want to thank the students, friends and co-students who tested and provided feedback on the tool.

A special thanks to the student in the study who suggested naming the tool Uncoil. Python syntax, like a Pythonsnake, has a strong hold on students when transferring to another language, making Uncoil an ideal name for the tool.

`Jenny Strømmen, June 1, 2022`

Contents

List of Figures

List of Tables

Acronyms

BIT OP Bitwise Operator Error.

EQ OP Equals Operator Error.

FINO INIT Field With No Initialiser.

IFNO BRKT If No Brackets Error.

IFW SEM Semicolon After If Error.

IGN RET Ignoring Return Error.

INT DIV Integer Division Error.

NOEQ METH No Equals Method Error.

NOSUP METH Forget To Call Supermethod In Subclass.

NO OVRD Forget to annotate with @Override.

ST OB Static On Object Error.

API Application Programming Interface.

AST Abstract Syntax Tree.

CE Compilation Error.

ERE Explicit Runtime Error.

IDE Integrated Development Environment.

IRE Implicit Runtime Error.

JVM Java Virtual Machine.

MVP Minimal Viable Product.

NST	Negative semantics transfer.
PaaS	Platform as a Service.
PVM	Python Virtual Machine.
SaaS	Software as a Service.
TDD	Test Driven Development.

Chapter 1

Introduction

This thesis aims to shed light on the semantic errors with no preexisting error messages in Java that we believe students who know Python will be vulnerable to. We investigate how we can develop a tool, Uncoil, to detect these errors and if our tool can help the students.

1.1 Two critical bugs as motivation

In a programming language, we have both *syntax*: the form of the language and *semantics*: the meaning of the language [1]. A *bug* is, as the preliminary quote states, when something goes wrong with a computer or a program. Different errors can cause bugs, but in this thesis, we focus on bugs caused by *semantic errors*. Semantic errors can go unnoticed because the syntax is valid [2]. Two examples of semantic errors that were not found before it was too late are the `goto fail;` bug by Apple [3] that caused a security issue and the Chrome OS bug [4] that caused the users unable to log in after an update. Even though these errors were not written in Java but C and C++, we will see that the same errors can be a problem in Java.

1.2 Context and Approach

The term *semantic transfer* is about transferring the semantics from the language we know to the language we are learning [5]. This thesis focuses on when semantic transfer has a negative outcome so we will use the term Negative semantics transfer (NST) found in [6]. *True Carryover Concept*, *False Carryover Concept* and *Abstract True Carryover Concepts* are proposed by [6], where NST would be a False Carryover Concept. *Mind shift theory* have been proposed by [7], where a *changed concept* relates to NST. However, for the scope of this thesis, we found the term NST to be most suitable.

How students and programmers transfer their knowledge from one programming language to another has been researched to some extent [6], [8], [9], [10], [11]. Similarly, the field of what errors Java students make have been studied [2], [12], [13], [14], [15], [16], including the semantic errors [2], [12], [15]. However, little research has been devoted to the crossover of these fields where we try to find what semantic errors students make when learning a new programming language, if they need help with them and how to help them. At the University of Bergen, students in informatics are introduced to Python in their first semester when they take the course INF100[1]. In the succeeding semester, most students will have INF101[2], a Java-based course. The students of INF101 are recommended to have INF100 before INF101. Therefore we believe that these students are good candidates for researching this field.

We know from previous research that students transfer their semantics when learning a new programming language [6], [8], and a recent study [17] also suggests this is the case for experienced developers. In this thesis it has been collected semantic errors that are believed to be relevant to a novice Java student that knows Python before, especially focusing on errors that might be caused by NST and does not have an existing error message.

Several papers [2], [12], [13], [14], [15], [16] try to find out what errors Java students make and some [2], [18], [19] have developed tools to help students overcome them. The main impression is that syntax errors are the most frequent errors, but semantic errors are more troublesome for the students. Similarly the papers [6], [8], [9], [10], [17] studies how students and programmers transfers their knowledge when going from one language to another, and states that NST indeed can be a problem. Some tools [11], [20], [21], [22], have been made as an attempt to help students and programmers with these transfers. However, only one [20] has focused on the transition from Python to Java. There has been found no papers suggesting to create error messages tailored to transfer from Python to Java or try to uncover how the students solve these errors in Java.

We have conducted a literature study, focus group and a mixed method study to find relevant errors. Moreover, we discuss how and if these errors should be automated. By letting seven students evaluate Uncoil and try to solve relevant errors we aim to gain knowledge on how to help them. We present the data from the evaluation and our tool is open source for further research.

[1]https://www.uib.no/en/course/INF100
[2]https://www.uib.no/en/course/INF101

1.3 Problem Description

There are many different programming languages, each with its own syntax. A quick look at some differences in the syntax in Python and Java would be

Python

```python
print("Hello world!");
```

```python
def function():
    return "Hello world!"
```

```python
for i in range(0, 5):
    print("Hello world!")
```

Java

```java
System.out.println("Hello world!"
    );
```

```java
public String function() {
    return "Hello world!";
}
```

```java
for (int i = 0; i < 5; i++) {
    System.out.println("Hello
        world!");
}
```

The syntax between Java and Python are different in the examples above, but their semantics are the same. Based on previous research listed in section 1.2 we assume that students and programmers use their knowledge from a previous language when learning a new one. This can have both benefits and be prone to error, the latter because there are cases where the syntax across two languages are similar, but the semantics can be different [23]. For example, transferring from Python to Java, the student can make assumptions that the body in an if-sentence is defined by *indentation* and not *curly brackets*. Listing 1.1 and Listing 1.2 illustrates this: The Java program would have the correct syntax, but unexpected semantics: the `method2();` statement would be called regardless of the if-sentence. This case serves as an example of Negative semantics transfer (NST) between two languages, where we expect the similar syntax to behave the same way, but it is different. The mistake in Listing 1.2 is the same problem that caused the `goto fail;` bug mentioned in section 1.1.

Python

```python
if (someCondition):
    method1()
    method2()
```

Listing 1.1: Python if-statement with no error

Java

```java
if (someCondition)
    method1();
    method2();
```

Listing 1.2: Java if-statement with error

Various checks take place before a program can be executed, and some of these checks catch different syntax errors and semantic errors. However, the Java code in Listing 1.2 has a semantic error that is not checked, so the student will not know anything is wrong. This can cause the error to persist for a long time according to [15], and as we know from the `goto fail;` bug mentioned in section 1.1, it might not be caught before it is too late.

It is important to point out that the errors portrayed in this thesis are not technically errors, because it is legal Java syntax and you may want the program to behave in that specific way. For this thesis, we focus on how students most likely will misinterpret the semantics of the syntax, and for that matter, we will call it an error.

1.4 Research questions

The questions this thesis aims to answer are

RQ1: *What semantic errors do students need help with when transferring from Python to Java?*

RQ2: *How can we develop a tool to automate feedback for semantic errors when transferring from Python to Java?*

RQ3: *How can such a tool help students when transferring from Python to Java?*

1.5 Research method

We have conducted a literature review in section 3.2, focus group [24] in section 3.3 and a mixed method study [25] in chapter 6 to get closer to an answer to RQ1, RQ2 and RQ3. To answer RQ1, we use knowledge from the literature review, focus group and mixed method study. To answer RQ2, we first look at how existing environments handle the errors and review previous research on how to represent an error message. Then, we demonstrate how the errors can be detected at an abstract level in chapter 4 before we present a concrete solution in chapter 5: *Uncoil*. To answer RQ3, we let students evaluate Uncoil in the mixed method study.

By reviewing existing literature, we can find errors that are thought to be relevant to research in this thesis. To limit the scope of this thesis, we have set three criteria in section 3.1 for including an error and included a maximum of six errors. By excluding papers that do not include semantic errors, we hope to learn more about these errors. If an error is mentioned as a problem across different studies, we will get a stronger indication that it should be researched further. By presenting the errors found in the literature review to a focus group [24], we can indicate if the errors we have chosen to include are relevant, getting closer to RQ1. We also set as a goal to discover new relevant errors in the focus group by letting participants suggest errors. We included this in the recruitment letter (Appendix H) such that they could think about relevant errors prior to the focus group. To get different viewpoints and create a fruitful discussion, we focused on recruiting different roles: lecturers, teaching assistants and students. By discussing the errors after showing examples of them, we expect to get closer to the following questions: Is the error relevant to novice Java students? Is the error more relevant when knowing Python before? By knowing what errors are relevant we get closer to an answer to RQ1, and by comparing the results from the literature review and focus group we get more confidence in our results.

As a preliminary step to research RQ2, we discuss if the errors should be automated and find existing feedback for the errors. To answer RQ2, we empirically test if the errors can be detected using different analysis methods. We look at previous research on creating error messages to extend our knowledge base on how to present an error message to the novice student. Further, we demonstrate how errors can be found at an abstract and concrete level. By creating a tool, Uncoil, to detect the errors, we give a proof of concept that these error messages can be automated in a way that we believe will be helpful to novice Java students that know Python.

We have chosen to use a mixed method study with an explanatory sequential design [25] to answer RQ1 and RQ3. By using an explanatory sequential design [25], with the qualitative questions acting as a follow-up to the quantitative questions, we hope to get a more reliable and informative result. By letting students solve the errors while filling out a survey, we hope to determine, together with the literature review and focus group, if the errors are relevant to the students and answer RQ1. We aim to get a better understanding of why the errors are a problem to the students by using the explanatory sequential study design [25], letting the qualitative answers explain why the students found the errors hard to solve. By analysing the qualitative answers on why an error was challenging to solve, we categorise the answers to find out if any may be caused by NST, bringing us closer to an answer to RQ1. This also partly answers RQ3; by categorising the answers, we can get an idea of what the students need help with and how Uncoil can help them. Letting the students try to solve the tasks and use Uncoil if they need help, we aim to answer RQ3. The explanatory sequential design enables us to tell how Uncoil can help the students by asking them why it helped (or not) as a follow-up question to whether Uncoil helped them.

By combining the mixed method study's results we hope to get reliable results, but the study has some limitations. The students were unsupervised, so we can not know whether Uncoil or some external factor helped them solve the error. The study is limited to the students' ability to express themselves in writing, so we allowed them to answer in either English or Norwegian. While the study design is based on being explanatory, we can not know the time lapse between the student's answers, so the student might have forgotten the reason why or given an incorrect answer. We asked the students to fill in the survey while doing the different tasks to keep the task fresh in their memory. Even though the students were unsupervised, which gives some limitations to the study, we believe the study benefits from letting the students solve the tasks in a known setting. Additionally, the study was anonymous to get honest answers from the students.

By combining the results from the literature study, focus group and mixed method study, we hope to get a more confident result for RQ1. However, the number of errors investigated is limited due to time constraints on this thesis, and there may be other relevant errors not discovered in this thesis. We hope that the focus group will help with this issue, collecting errors suggested by the participants. Even though we have found one way to automate the errors to answer RQ2, there may be other ways to accomplish this, discussed further in section 4.6 and chapter 8.

1.6 Contribution

We deliver a small data set from the focus group about the errors found in this thesis. We also have a data set from the mixed method, evaluating both the errors found in this thesis and Uncoil, found in Appendix B. Most of the answers are given in Norwegian. Furthermore, we provide a list of errors found in Appendix F, based on previous research. We present the list of errors found to be relevant in this thesis that needs to be focused on in Table 6.5.

The work in this thesis is novel for these reasons:

- We have found no papers on what errors novice students makes when they transfer from Python and Java.
- We found only one paper [18] focusing on semantic errors, an none that focuses especially on error messages that do not provide an existing error message.
- Few tools, only one [20] to our knowledge, try to correct the misconceptions when going from Python to Java.
- Earlier studies mentioned in section 1.2 try to find if the students have trouble when transferring to another language. This thesis attempts to help them with these struggles, and how the students think when trying to solve these errors.
- Similar tools [20], [22] to Uncoil have been evaluated by doing a pre and post-test, but in this thesis, we present a different evaluation method for such tools to gain insight into how the students solve the errors.

1.7 Outline

The rest of this thesis is structured as follows:

Chapter 2
Explains the theory behind this thesis. We go through the key differences between Python and Java that are important to know for this thesis, how we can analyse code for errors and background needed to understand Uncoil. Additionally, we define what we mean by an error in this thesis.

Chapter 3
This chapter aims to find relevant semantic errors for novice Java students as a preliminary study for the mixed method study and discovering errors Uncoil should detect. We

conduct both a literature review and a focus group to achieve this.

Chapter 4
We uncover missing feedback from existing environments that we believe is crucial to novice Java students, and further demonstrate how to produce this feedback. The abstract approach for finding the errors we present in this chapter are later used to develop Uncoil in chapter 5, and partly answers RQ2.

Chapter 5
This chapter goes through how Uncoil is implemented by using the knowledge from chapter 4, giving a more concrete answer to RQ2. We also discuss the methods used to implement Uncoil.

Chapter 6
Together with chapter 3, we aim to answer RQ1 and RQ3. Here we present a mixed method study with seven participants to learn what errors are a problem to the students and if Uncoil can help them. We try to answer if the error messages are needed and useful.

Chapter 7
We discuss the results for the research questions in this thesis and the limitations of our studies.

Chapter 8
We compare Uncoil and our research methods to existing literature.

Chapter 9
We propose further work and development of Uncoil.

Chapter 10
We present the conclusions of this thesis.

Chapter 2

Background

This chapter defines what we mean by a semantic error, compares Python and Java and describes how to analyse code.

2.1 Definition of an *error* in this thesis

First, let's define what we mean by an *error* in this thesis. As noted in section 1.1, a programming language has syntax and semantics. A *syntactic error* is caused by a program having the wrong form, so the program can not be executed, while a *semantic error* is caused by having a wrong meaning and may stop the program from executing or give a wrong answer [12]. A *logical error* appears when the programmer has misinterpreted the task [12].

We call the errors in this thesis semantic errors based on the perspective of a novice Java student that makes assumptions from Python syntax. Specifically, the student write correct syntax for Java, but gives it the wrong semantics. In contrast to a type error as we will explain in subsection 2.3.3, the errors in this thesis do not give any feedback to the programmer.

2.2 Programming languages

This section points out some significant differences between Python and Java for this thesis. Before diving into the differences between Python and Java, we need to know some basics about programming languages, compilers and interpreters. This is explained in the succeeding subsections.

2.2.1 Why we need different programming languages

For humans and computers to be able to communicate with each other, they need a common language: *programming languages* [1]. The problem depicted earlier in section 1.3 would probably not be an issue if we only had one programming language. So why do we need different programming languages? We need different languages for different purposes. For example, Python has an easy syntax and is used by beginners in programming and scientists [26]. ANTLR (ANother Tool for Language Recognition) is a *metalanguage*, used to define other programming languages [27]. JSON (JavaScript Object Notation) is an *exchange language* used to communicate between different programs [28].

2.2.2 Introduction to compilers and interpreters

Compiler

Even though we stated earlier that the programmer and the machine speak the same language with a programming language, this is a truth with some modifications. A machine does not understand the program as we write it in a natural language, but understands *machine language* that consistist of 0 and 1s. In the 1940s, programmers had to write programs in machine language, making larger programs very complex. [23]

Assembly language is easier to understand for humans and converts easily to machine code. The language gives an abbreviation for the corresponding machine instruction, and translating the abbreviation to machine code was done by an *assembler*. Nowadays, we can write high-level instructions like a Java program. A compiler is a program that takes source code as input and gives machine code as output, and will translate the Java code to assembly code or machine code for us. When the operating system understands the code, we call it *object code*. The compiler can also return machine code for a *virtual machine*, for example *bytecode* for Java Virtual Machine (JVM). Bytecode is similar to assembly code but for Python and Java. [23]

Interpreter

Interpreters execute the code ”on the fly” without compiling everything to machine code beforehand. An example of an interpreter is the JVM that executes Java bytecode. Java has Java bytecode that can be executed by the JVM while Python has Python bytecode that can be executed by Python Virtual Machine (PVM). [23]

2.2.3 Python vs. Java

This section collects the differences between Python and Java that are important for this thesis.

Compiled vs interpreted

Python is called an *interpreted* language, while Java is called a *compiled* language [23]. Still, as we see in Figure 2.1, they essentially do the same thing [29], [30, §1.2]: The source code is compiled to bytecode before the bytecode is interpreted on the Virtual Machine.

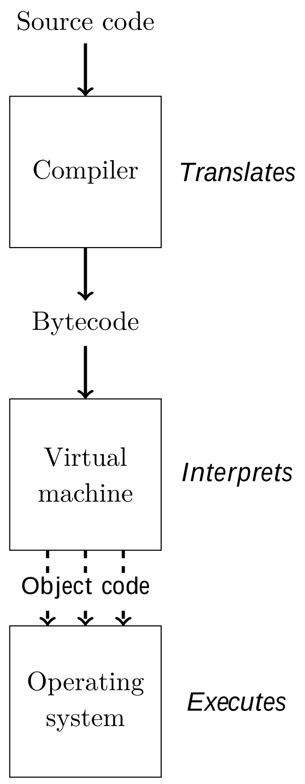


Figure 2.1: Illustrates how Java and Python are compiled and interpreted. First, the compiler takes in the source code and compiles it to Java or Python bytecode. Then the virtual machine translates the program to object code and executes it on the operating system line by line. Each dotted line is supposed to be a line that is being executed. Inspired by [23, pp. 17-20, Example 1.9].

Because of the virtual machines, both Java code and Python code are platform independent: The virtual machine is a virtual environment that interprets bytecode to machine dependent code. [30, §1.2]

Type system

According to Scott [23], a *type* can be described as the set of values a variable is allowed to have. For example, we may have a variable, `a` that should only be allowed *integer* values. Then we need to set the type for `a` to `integer`. Lämmel [31] states that a programming language can choose to declare types in the syntax. For example in Java we have to declare the types in the syntax, but this is not the case in Python. Further, he explains that Java has a *static type system* while Python has a *dynamic type system*. The first means that the types of variables are known before you run the program, and the compiler can check that the types match. The latter means that the types are checked during the program's execution, at run time.

Strongly typed languages does not allow type conversion while *weakly typed* languages do, and both Java and Python are thought of as strongly typed languages [23]. Still, there are examples where they are weakly typed. In Python, the code `print("python"+3)` results in an error. However, Java does a string conversion on the integer `3` behind the scenes [32, §5.1.11] when adding `"python"+3`, so in this scenario, Java is weakly typed. When performing a division on two integers like `7/5`, Java returns an integer [32, §15.17.2] and is strongly typed. Starting from Python version 3, Python is weakly typed in the same scenario because the division operator / returns a decimal [33].

According to Khoirom *et. al.* [26], when we are *casting* a variable we are specifying its type. They explain how to cast to types in Java and Python as follows: In Python this can be done in two ways: `a = int (1.8)`, telling Python that it should round `a` to the integer `1`. The other way is to use a function, for example `s = str(a)`, converting the integer `a` to a string. Further, they note that we have two types of type casting in Java: implicit or explicit. The implicit type casting works when *widening* a type, for example when going from `byte` to `short`: `byte a = 50; short b = a;`. Explicit casting is needed when we want to *narrow* a type, for example from `double` to `int`: `double a = 1.8; int b = (int) a;`. Another thing to consider in Java is also the risk of *overflow* of numbers [32, §4.2]. This is not a problem in Python, because it allocates bits necessary for the number dynamically [34, §3.2].

Equals method and equals operator

The *equals method* in a class in both Python and Java defines how to tell if two objects are equal to each other. According to the documentations for Java [32, §4.3.2] and Python [34, §6.10], in both languages the method has the default implementation of comparing two objects using their memory location but we can implement a custom equals method for a class.

As found in the documentations for Python [34, §6.10] and Java [32, §15.21], the *equals operator* `==`, is used for comparing elements both in Python and Java. However, the semantics can be different when using the operator across the two languages, depending on the type the operator is comparing. When comparing *primitive types* in Java, we use the equals operator because the types are built in the language and are not objects [32, §4.2]. However, using the equals operator on objects compares memory location, so we need to use the equals method [32, §4.3.2]. In Python, objects that override the equals method can be compared using the equals operator because it calls the equals method of the class, while the `is` operator compares references [34, §6.10]. Java arrays inherit the equals method from Object, so we need to use `Arrays.equals(arr1, arr2)` to compare the content of two arrays [32, §4.3.2].

Indentation vs brackets

Python uses *indentation* to group parts of the code [34, §2.1.8], for example the body of an if-statement illustrated in Listing 2.1. Consequently, Listing 2.1 and Listing 2.2 will behave differently. The `method2()` in Listing 2.2 will not be in the if body, and executed each time the code runs.

```
if (someCondition):
    method1()
    method2()
```

Listing 2.1: Python if-statement with two children

```
if (someCondition):
    method1()
method2()
```

Listing 2.2: Python if-statement with one statement and a sibling

Java does not use indentation to group parts of the code, but *braces*: {} [32, §14]. This means that the code in Listing 2.3 and Listing 2.4 will behave differently. In the first Listing 2.3, the `method2();` would be called each time, and has the same semantics as Listing 2.2. In the second Listing 2.4 `method2();` would be included in the if-statement because of the brackets.

```
if (someCondition)
    method1();
    method2();
```

Listing 2.3: Java if-statement with one child and one sibling with indentation

```
if (someCondition) {
    method1();
    method2();
}
```

Listing 2.4: Java if-statement with two children.

We can see a syntax similarity between the first Python Listing 2.1 and the first Java Listing 2.3, yet they have a very different outcome, which supports [23] that says syntax can be similar between two programming languages but have different semantics.

Python

```
if (someCondition):
    method1()
    method2()
```

Listing 2.5: Python if-statement with two children

Java

```
if (someCondition)
    method1();
    method2();
```

Listing 2.6: Java if-statement with one child

Python uses indentation, so it is clear what if-statement an else-clause belongs to [34, §2.1.8]. In contrast, Java can have a *dangling else* [32, §14] making this harder to determine. It would not be a problem if we used brackets to define the body. An example of this is shown in Listing 2.7

```
if (someCondition)
    if (someOtherCondition)
        doSomething();
```

```
else doSomethingElse();
```

Listing 2.7: Java example of dangling else problem

Semicolon and boolean operators

For now, it suffices to know that a *statement* is a piece of code that tells the program what to do and that an *expression* is a piece of code that evaluates to a value [35]. Python uses a new line to end statements unless it is on the same line as other statements [34, §2.1]. When using Java, a statement is finished with a semicolon and would cause a syntax error if the semicolon was missing [32, §14]. The *boolean operators* are operators that lets you evaluate boolean expressions, like *true* and *false*. The AND operators are && in Java while and in Python, and the OR operators are || in Java while or in Python [32, §4.2.5], [34, §6.11].

2.3 Source code analysis

We have learnt from subsection 2.2.2 that a compiler takes in source code and gives machine code as output. We can also use the compiler techniques to analyse the source code, and the first steps of the compiler's job are to find out the meaning of the program: *semantic analysis* [23], which is what we want to do. Therefore in this section, we go through the steps to be able to do semantic analysis.

2.3.1 Lexical analysis

Lexical analysis organises the source code into *tokens*, the smallest meaningful unit of a program, for further handling [23]. For example, the Java program:

```
class A {}
```

Listing 2.8: Java program with an empty class

is read as:

```
c,l,a,s,s,␣,A,␣,{,}
```

Listing 2.9: Java program characters

and the tokens are grouped together as showed in Listing 2.10.

```
"class", "A", "{", "}"
```

Listing 2.10: Java program tokens

Normally the spaces are ignored. The token "class" is a *keyword* in Java. The class-name "A" is an *identifier*, and the "{" and "}" are *symbols*.

2.3.2 Syntax analysis

Syntax analysis is done by a *parser* that checks if the tokens produced by the lexical analysis constructs a valid program, and outputs the source code in a more analyse-friendly format: a *tree structure*. [23]

Trees in a software context

Trees in a software context are, among other things, a convenient way to represent source code. Based on [36], we describe a tree as follows: A tree consists of *nodes*, the top node is the *root* of the tree, and the end nodes are *leafs*. Additionally, two nodes have the same *parent* they are *siblings*. At last, nodes above the parent are *ancestors* to the node, while nodes below the child(ren) are *decedents* of the node. Figure 2.2 illustrates the nodes in a tree.

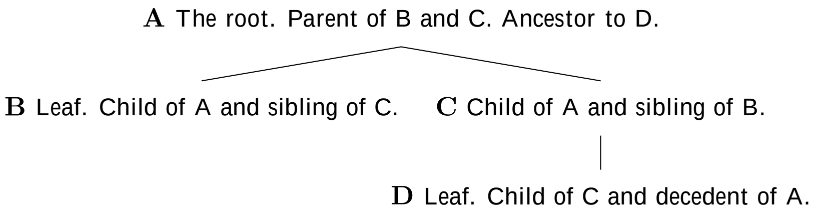

A The root. Parent of B and C. Ancestor to D.
B Leaf. Child of A and sibling of C.
C Child of A and sibling of B.
D Leaf. Child of C and decedent of A.

Figure 2.2: A simple tree with description of the nodes.

Parsing and grammar

When a parser creates a tree, it is called a *parse tree* or *concrete syntax tree*, and is defined by the *grammar* of the language [23]. Listing 2.11 provides a simplified grammar for a statement in Java, and is inspired by [23, p. 29, Example 1.22] and [32, §19]. The arrow → can be read as *can be*, and the vertical lines | can be read as *or*. We can see from the grammar in Listing 2.11 that a statement can consist of other statements or *expressions*. The expression in an if-statement would in Listing 2.11 evaluate to a boolean value.

A statement in the grammar in Listing 2.11 can be an `if-statement`, `expression`, `blockstatement`, or an `empty statement: ;`. In this grammar the lowercase words and symbols are tokens, also called *terminals*, while the *non-terminals* are marked with uppercase letters [23]. We can see that an `if-statement` only allows one statement and the condition is an expression that evaluates to a value. We also see that a statement can be a `blockstatement`, that can hold several statements. For brevity the definition of an `expression` in the grammar has been left out in Listing 2.11.

```
STATEMENT       -> if ( EXPRESSION ) STATEMENT
                 | EXPRESSION ;
                 | BLOCKSTATEMENT
                 | ;
BLOCKSTATEMENT -> { STATEMENTS  }
STATEMENTS      -> STATEMENT STATEMENTS | STATEMENT
```

Listing 2.11: Simplified Java grammar for a statement

Parse trees

If our code does not agree with the rules of the grammar, the parser will return a *syntax error* [23], and as mentioned in section 2.1, the program will not execute. For the grammar in Listing 2.11 an example of illegal code can be found in Listing 2.12.

```
if {someCondition} doSomething();
```

Listing 2.12: Simple syntax mistake in Java

Listing 2.12 will give a syntax error because the brackets should be parenthesis according to the grammar in Listing 2.11. If we replace the brackets with parenthesis, the parser can construct the parse tree illustrated in Figure 2.3.

```
if (someCondition) doSomething();
```

Listing 2.13: Simple Java program that matches grammar

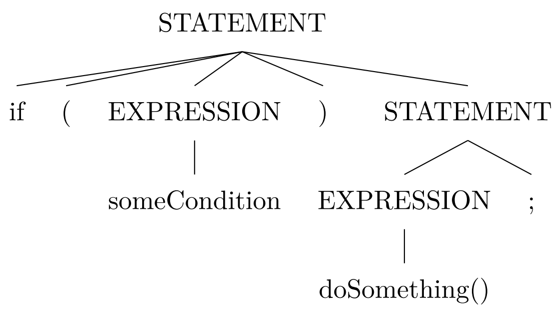


Figure 2.3: Parse tree of Listing 2.13 from grammar Listing 2.11

In a parse tree, the root is the `program`, and the leaves are the tokens. We would get the original source code if we collected all the leaves left to right in the parse tree.

2.3.3 Semantic analysis

For the compiler to output an equivalent machine code to the source code, it needs to know the meaning of the source code. After the parser has created the parse tree, the semantic analyser can start to analyse the tree. Depending on the language being analysed, different checks can take place at the semantic analysis stage. There are two types of semantic analysis: ***static analysis*** and ***dynamic analysis***. We know from section 2.2.3 that Python uses dynamic types, so types are checked and decided during execution, giving a dynamic analysis. In contrast, Java uses static typing and can do most type checking at compile time, giving a static analysis. Consequently, in this thesis, we focus on static analysis. [23]

The semantic analyser uses the parse tree from the syntax analysis to create an ***Abstract Syntax Tree (AST)*** to analyse the code. Abstract Syntax Tree (AST) are parse trees without unnecessary nodes, for example the parenthesis, and the AST for the parse tree in Figure 2.3 is illustrated in Figure 2.4. After analysing the tree, the semantic analyser gives the AST as output to the ***back-end*** of the compiler, where the transformation to machine code happens. [23]

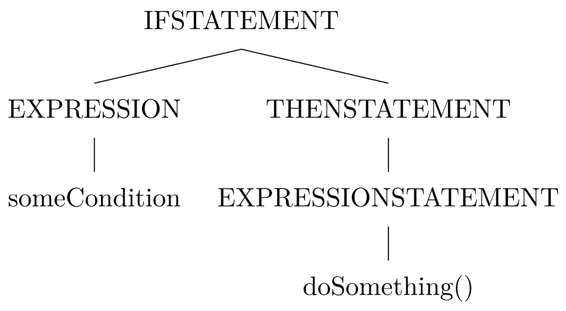


Figure 2.4: Abstract syntax tree from parse tree in Figure 2.3

For the compiler to generate the correct output, the analyser needs to know the semantics of each node, an an AST with the semantics of the nodes attached is said to be ***decorated*** or ***annotated*** [23]. The compiler already knows the semantics of some nodes, for example, the keyword `if` [37]. But how can it know the semantics of the variable `someCondition` in Figure 2.4? The programmer defines the semantics for the variable name: where it is placed in the code, the type of the variable and the variable's value [37]. The compiler keeps track of the semantics by using a ***symbol table***, with the variable name as key and the semantics attributes as values [23]. For example, the condition in Listing 2.13 should be of type boolean. We could check this by registering the declaration of the variable `someCondition` in a table

{`someCondition: boolean`},

where the key would be the name of the variable, and the value would be the type of the variable. To check that `someCondition` is `boolean` when used in an if-condition, we can do a lookup in the table. If the type of the `someCondition` is a `String`, we know we have a type error. This is an example of a semantic error because the syntax is correct, but the semantics are wrong. Since the compiler does type checking in Java, this error would generate an error message from the compiler. [23]

2.3.4 Different ways to analyse the source code

We can, as discussed in subsection 2.3.3, use the techniques of the front-end of the compiler to get the annotated AST and perform analysis on it. For Java, we can also compile the file to bytecode and analyse the bytecode [38]. We will not go into how to analyse bytecode in this thesis, but SpotBugs[1] is an example of a tool that uses bytecode to analyse Java files for bugs. Section 4.3 shows an example of why bytecode is not used to analyse the errors in this thesis.

To analyse the annotated AST, we could use the *visitor pattern*. The visitor pattern is a design pattern used to visit the nodes in a tree. The root node *accepts* a visitor, that for each type of node has an implementation of the work it should do for this type of node. When you visit an AST, you start at the root and visit each child until you reach the leaves. [39]

For example, if you want to print every method invocation, you define the method in Listing 2.14 that prints the `methodInvocation`.

```java
public void visit(MethodInvocation methodInvocation) {
    System.out.println(methodInvocation);
}
```

Listing 2.14: Java visit method example

In this thesis it is used *JavaParser* to analyse code. JavaParser lets you inherit from a simple visitor, the `VoidVisitorAdapter` class, to traverse the tree. It visits the children in random order. [40]

[1]https://spotbugs.github.io/

Chapter 3

Discovering relevant semantic errors for novice Java students

In this chapter, we try to get closer to an answer to

RQ1: *What semantic errors do students need help with when transferring from Python to Java?*

We collect and discuss the errors studied in this thesis to contribute to the answer to RQ1. Some of the errors are found through a literature review, while others are found during a focus group. Some of the errors were found in the literature review and presented to the focus group to see if they would be relevant or not for this study.

First, we define what criteria an error needs to pass to be included in this thesis, then we present the literature review and discuss the errors found. The rest of the chapter shows the focus group and concludes on which errors are studied further.

3.1 Criteria for including an error in this thesis

We have decided to include a maximum of six errors to limit the scope of this thesis and set three criteria for an error to be studied:

1. The error should not give an Explicit Runtime Error (ERE) or a Compilation Error (CE).
2. The error should be detectable by static analysis.
3. The error should be detectable by analysing the file it is present in.

A CE is an error that is caught by the compiler and gives an error message to the programmer. As we have seen in subsection 2.3.3, this can be both syntax errors and semantic errors like type errors. Therefore, we need to exclude the semantic errors that already have an error message. By ERE, we mean that the program compiles but has an error telling explicitly what the problem is when executed. For example, an `ArrayIndexOutOfBoundsException` in Java will tell what the problem is: the index is too large for the array being indexed. This thesis tries to catch the errors that do not already have an error message, so the criteria are defined to exclude the explicit error messages from the study. The second criterion is to ensure that we can reasonably detect the error. The third criterion is to limit the requirements to Uncoil.

3.2 Literature review

This literature review focuses on collecting previous research on what semantic errors students make to see if they can be relevant for this thesis. A paper had to include semantic errors or classify what type of errors the students have problems with to be included in the study. The errors found in the reviews were reconstructed and tested empirically to see if they fit the criteria. All the errors were compiled with a 16.0.1.hs-adpt compiler.

Most of the errors found in the literature review did not meet our criteria, and some errors were too unclear to reproduce. The reproduction of the errors can be found in table F.1, Appendix F. The results for errors that meet the first criteria can be found in Table G.1, Appendix G. The second criteria for the relevant errors are shown in chapter 4.

From a total of five relevant papers, six semantic errors that could be relevant for a Java beginners student have been found. Two of the errors were discarded after discussing them in the focus group. In this section, we present the papers and briefly explain the errors.

To find out what errors students in Java make, Hristova *et. al.* [2] collected sixty-two errors students make in Java, and presents twenty of them in their paper. The errors they found are based on interviews of professors and teaching assistants, and are listed in Appendix D. Out of all the errors they presented, five of them are investigated further: Equals Operator Error $\binom{\text{EQ}}{\text{OP}}$, Bitwise Operator Error $\binom{\text{BIT}}{\text{OP}}$, Semicolon After If Error $\binom{\text{IFW}}{\text{SEM}}$, Static On Object Error $\binom{\text{ST}}{\text{OB}}$ and Ignoring Return Error $\binom{\text{IGN}}{\text{RET}}$. An example of the errors and what they called the error is presented in Table 3.1.

Error	Hristova *et. al.* [2] error description	Example

$\frac{EQ}{OP}$	== versus .equals (faulty string comparisons)	`Object a1 = new Object();`
		`Object a2 = new Object();`
		`boolean condition = a1 == a2;`

$\frac{BIT}{OP}$	&& vs. & and || vs. |	`boolean condition = true & false;`
$\frac{IFW}{SEM}$	incorrect semicolon after an if selection structure before the if statement or after the for or while repetition structure before the respective for or while loop	`if (condition) ; {method1();}`

$\frac{ST}{OB}$	invoking class method on object	`Demo demo = new Demo();`
		`demo.staticMethod();`

$\frac{IGN}{RET}$	invoking a non-void method in a statement that requires a return value	`"abc".toUpperCase();`

Table 3.1: Examples of errors collected from Hristova *et. al.* [2]'s study.

$\frac{EQ}{OP}$ is an error where two objects are being compared by using the equals operator. In Python this operator will use the defined equals method on the objects. In Java, it will check the references for the object to decide its equality. The error $\frac{BIT}{OP}$ happens when the bitwise operators & or | are confused with the short-circuit logical operators in Java && and ||. The $\frac{BIT}{OP}$ was the cause of the previously discussed Chrome OS bug as mentioned in section 1.1. The $\frac{IFW}{SEM}$ occurs when there is a semicolon right after the if-condition. $\frac{ST}{OB}$ takes place when we call a static method on an object. The $\frac{IGN}{RET}$ happens when we call a method without storing it in a variable.

To analyse the errors proposed by Hristova *et. al.* [2], Brown and Altadmri [15] collected data from students using BlueJ over two years. They stored how frequent an error appears and how long time it took to fix the error. They found that the errors $\substack{\text{IGN}\\\text{RET}}$, $\substack{\text{EQ}\\\text{OP}}$, $\substack{\text{BIT}\\\text{OP}}$, $\substack{\text{IFW}\\\text{SEM}}$ are the top four errors that take the longest time to solve for the students, even though they are not as frequent as syntax errors. They omitted $\substack{\text{ST}\\\text{OB}}$ in their study. The time to fix parameter had a max limit of 1000 seconds (about 15 minutes). The numbers from Brown and Altadmri [15] are presented in Table 3.2. Frequency ranking # is the rank of the error when sorting all the errors in their study by frequency.

Error	**Brown and Altadmri [15] error name**	**Frequency**	**Time to fix (median)**	**Frequency ranking #**
$\substack{\text{IGN}\\\text{RET}}$	N discSem	274963	1000	5
$\substack{\text{EQ}\\\text{OP}}$	B strEqSem	274387	1000	6
$\substack{\text{IFW}\\\text{SEM}}$	E smiConSyn	108717	387	10
$\substack{\text{BIT}\\\text{OP}}$	D andOrSyn	61965	1000	12

Table 3.2: The frequency and time to fix for $\substack{\text{IGN}\\\text{RET}}$, $\substack{\text{EQ}\\\text{OP}}$, $\substack{\text{BIT}\\\text{OP}}$ and $\substack{\text{IFW}\\\text{SEM}}$ according to Brown and Altadmri [15, Table 1].

To get a *total time to fix* for the errors, Brown and Altadmri [15] also calculates the `frequency` times the `mean time to fix`, giving a slightly different result than in Table 3.2. By plotting the values in Figure 3.1 we see that clearly the $\substack{\text{EQ}\\\text{OP}}$ and $\substack{\text{IGN}\\\text{RET}}$ are the most troublesome errors among the four errors.

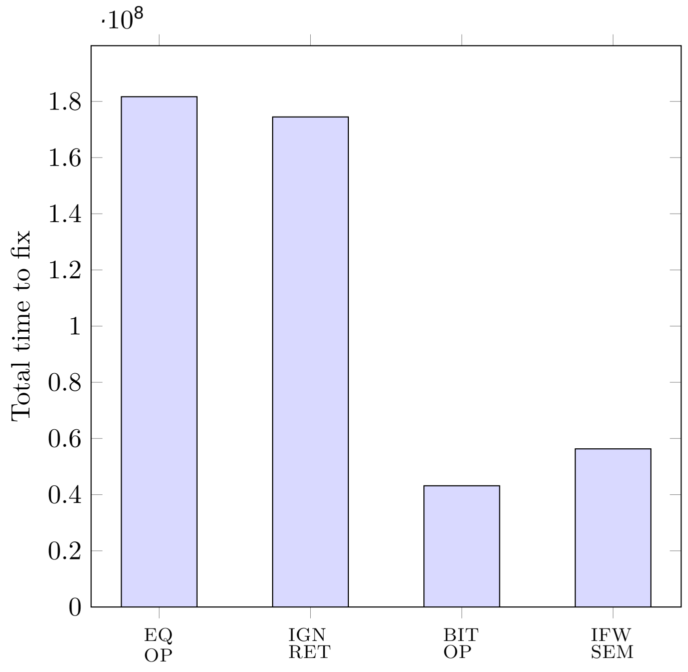


Figure 3.1: The total time to fix for the errors $\frac{\text{IGN}}{\text{RET}}$, $\frac{\text{EQ}}{\text{OP}}$, $\frac{\text{BIT}}{\text{OP}}$ and $\frac{\text{IFW}}{\text{SEM}}$ according to [15, Fig. 1].

Integer Division Error ($\frac{\text{INT}}{\text{DIV}}$) is mentioned by Tshukudu and Cutts [6] as a potential problem for students transferring from Python to Java: when dividing two integers in Java, it rounds down to the first integer, but when using Python 3 the operation will return a float number. They also find that students have problems with $\frac{\text{EQ}}{\text{OP}}$.

The errors Chan Mow [12] found by analysing errors made by students across three courses can be found in Appendix E. Chan Mow [12] also mentions a possible loss of precision as a logical error, specifically storing a double in type integer that could lead to a $\frac{\text{INT}}{\text{DIV}}$.

Rosbach [16], [41] classified errors novice students make, and found that the category *Wrong condition*, that can be caused by $\frac{\text{EQ}}{\text{OP}}$, is the most common among Java novice students.

3.2.1 Discussion

This section discusses the errors found in the literature review and how these can be problematic for a student when transferring from Python to Java. We also justify why we call the errors semantic errors.

$\frac{\textbf{EQ}}{\textbf{OP}}$. Looking at Figure 3.1, this is probably one of the most relevant errors when going from Python to Java. By Hristova *et. al.* [2], $\frac{\text{EQ}}{\text{OP}}$ is classified as a syntax error while

Brown and Altadmri [15] classifies this as a semantic error. Here we agree with Brown and Altadmri [15] that this is a semantic error because the program is syntactically correct. $\substack{\text{EQ}\\ \text{OP}}$ is specified by Hristova *et. al.* [2] as a potential error when making string comparisons. In contrast, Tshukudu and Cutts [6] finds that students believe the equals operator will behave the same in both Python and Java when comparing arrays, while we know from section 2.2.3 that it does not. We believe that the students struggle with $\substack{\text{EQ}\\ \text{OP}}$ for all objects. $\substack{\text{EQ}\\ \text{OP}}$ is also classified by Rosbach [16] as a problem with conditions, a common problem among the novice Java students according to his results.

$\substack{\textbf{BIT}\\ \textbf{OP}}$. Looking at Figure 3.1, we see that this is the error with the lowest score on total time to fix value among the four errors, suggesting that it may be not that relevant to the students. This is classified as a syntax error by Hristova *et. al.* [2] and Brown and Altadmri [15]. Still, we classify this as a semantic error in Java: Both the short circuit operator and the bitwise operator will give the correct syntax. We know from section 2.2.3 that Python use the keywords `and` and `or` to evaluate logical expressions, while Java uses `&&` and `||`. Even though the Figure 3.1 suggests that $\substack{\text{BIT}\\ \text{OP}}$ might not be that relevant, the Chrome OS bug mentioned in section 1.1 shows that the bug can be serious when it is present and should be taken into consideration.

$\substack{\textbf{IFW}\\ \textbf{SEM}}$. Figure 3.1 shows that $\substack{\text{IFW}\\ \text{SEM}}$ may be relevant to the students, even with a significantly lower score than $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{IGN}\\ \text{RET}}$. Python uses a colon after the if-condition to define the body. It is similar to the syntax if we put a semi-colon after the condition in Java, but it will not behave the same. We believe the novice Java students can be vulnerable to this error because of the matching syntax between Java and Python. We classify this as a semantic error while Hristova *et. al.* [2] and Brown and Altadmri [15] classifies this as a syntax error. Again we argue that this is a semantic error because the syntax is valid, yet the semantics are likely to not as intended. This error can also be true for other loop structures, but to limit the scope of this thesis, we only focus on the if-statement structure.

$\substack{\textbf{ST}\\ \textbf{OB}}$. This is described as a semantic error by Hristova *et. al.* [2], but Brown and Altadmri [15] does not include this error in their study. We agree with Hristova *et. al.* [2] that this is a semantic error. Calling a static method on an object would not be a problem in Python because the method call is decided at run time, but in Java the method call uses the method defined by the type on the object.

$\substack{\textbf{IGN}\\ \textbf{RET}}$. This also looks like a relevant error for the students when looking at Figure 3.1. The problem here is that the return value is ignored. This is described by a logical error by Hristova *et. al.* [2] while Brown and Altadmri [15] classifies this as a semantic error. We argue this is a semantic error because a logical error is when the programmer misinterprets the task. The syntax for $\substack{\text{IGN}\\ \text{RET}}$ in Java and Python is similar to each other, and it has the same semantics. Still, we included this error in the focus group to discuss if it could be relevant for the students when learning Java.

$\substack{\textbf{INT}\\ \textbf{DIV}}$. The Integer Division Error ($\substack{\text{INT}\\ \text{DIV}}$) was pointed out from Tshukudu and Cutts [6] as an error based on the transfer of semantics between Python and Java: integer division in Java and Python has similar syntax but different semantics. This study was found after the focus group, so $^{\text{INT}}_{\text{DIV}}$ was not presented to the group. However, $^{\text{INT}}_{\text{DIV}}$ is mentioned as a possible cause for the *improper casting* error in Hristova *et. al.* [2] and Chan Mow [12] mentions this as a logical error, further supporting that we should include $^{\text{INT}}_{\text{DIV}}$ in this thesis. $^{\text{INT}}_{\text{DIV}}$ is not included by Brown and Altadmri [15]. However, we argue this is a semantic error because to divide two integers in Java is legal syntax, but to expect a decimal from the division is a semantic error.

3.3 Focus group

This section aims to discuss the errors found in the literature review to see if they can be relevant to research in this thesis. Five of the semantic errors found in the literature review was presented to the focus group: $^{\text{EQ}}_{\text{OP}}$, $^{\text{BIT}}_{\text{OP}}$, $^{\text{IFW}}_{\text{SEM}}$, $^{\text{ST}}_{\text{OB}}$ and $^{\text{IGN}}_{\text{RET}}$. $^{\text{INT}}_{\text{DIV}}$ was not included because it was found after the focus group.

3.3.1 Method

By discussing the errors found in a focus group we can compare the results to the literature review, either strengthening or weakening if we should include an error in this thesis. After the group was presented with an error, they answered three questions on a poll about the error, aiming to know if the error was frequent, relevant to an INF101 student and relevant if the student know Python before. The participants could not see the answers while voting. After the participants had voted, we used the results to create a discussion. Data was collected by asking questions about the errors to the group, let the participants vote on a poll, and take notes from the discussion. At the end the participants were encouraged to propose errors they thought were relevant. Even though we collect quantitative data from the poll, we realise that the group is too small to do any statistical analysis. However, we found the data useful to create a discussion. The poll was mainly created to make the participants interact in the group and start a discussion, because the meeting setup had to be changed to a digital meeting due to Covid-19 restrictions.

Questions

The questions asked on the poll for each question were quantitative and are presented in Table 3.3.

	Question	**Answer type**
Q1	Have you seen any cases of this mistake among students or done it yourself?	Yes often, Yes, No, No answer
Q2	Is it likely that an INF101 student makes this mistake?	Yes, Maybe, No, No answer
Q3	Would a student that knows Python beforehand be more likely to make this mistake than a student who does not know Python?	Yes, Maybe, No, No answer

Table 3.3: Questions for focus group

In addition, as a qualitative method, it was also asked why the group voted as they did to create a discussion.

3.3.2 Recruitment

Both lecturers, teaching assistants and students were invited to get a representative group. The students and teaching assistants were recruited by sending out a recruitment letter via informal conversations in the study hall for informatics at the University of Bergen. The letter can be found in Appendix H and is written in Norwegian. The students were asked to send a mail if they were interested. For the lecturers, the recruitment happened via mail and informal conversations at the offices. The focus group had six participants: two lecturers in Java, one former teaching assistant in Java and a former lecturer in Python, one teaching assistant in Java and two students with both Python and Java backgrounds.

3.3.3 Setup

The focus group was held digitally due to Covid-19 restrictions. The presentation used in the group can be found in Appendix I.

3.3.4 Results

Figure 3.2 depicts how often the error has occurred either among students or themselves. We see that the group strongly agrees that $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{IGN}\\ \text{RET}}$ is a frequent error, while $\substack{\text{BIT}\\ \text{OP}}$, $\substack{\text{IFW}\\ \text{SEM}}$ and $\substack{\text{ST}\\ \text{OB}}$ are less frequent according to the group, with $\substack{\text{ST}\\ \text{OB}}$ being the least frequent error. Even though two answered `yes` on having observed $\substack{\text{IFW}\\ \text{SEM}}$ it was commented during the discussion that this was cases of for-loops or while-loops. It was mentioned that the $\substack{\text{BIT}\\ \text{OP}}$ was a normal mistake for beginners in Java to make. The group discussed that even though students struggle with static methods in Java, the $\substack{\text{ST}\\ \text{OB}}$ was rarely or never seen.

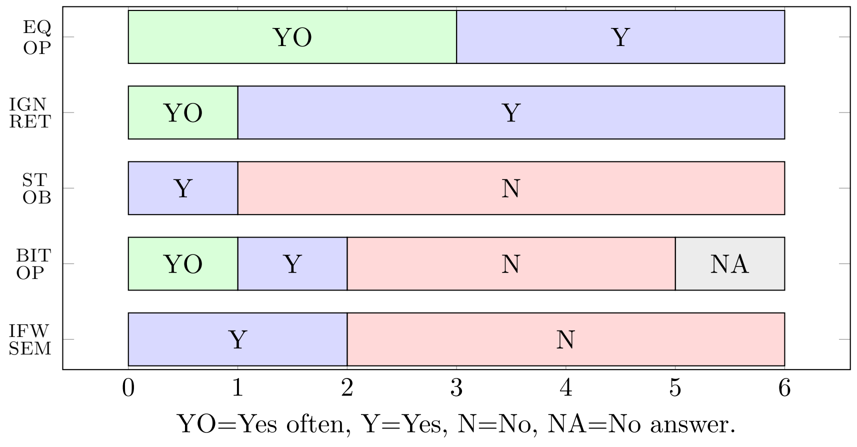


Figure 3.2: Focus group answer to Q1: "Have you seen this mistake among students or done it yourself?"

Figure 3.3 shows what the participants answered when asked if a beginners Java student would make these errors. Everyone in the group agreed that $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{IGN}\\ \text{RET}}$ is likely to be made by an INF101 student. The errors $\substack{\text{BIT}\\ \text{OP}}$ and $\substack{\text{IFW}\\ \text{SEM}}$ is less likely according to the group. The group answers that the $\substack{\text{ST}\\ \text{OB}}$ is the least likely relevant error for the students. During the discussion some in the group mentioned that regarding $\substack{\text{IFW}\\ \text{SEM}}$, the students might just place semicolons randomly until it compiles and that this might cause the error to be relevant.

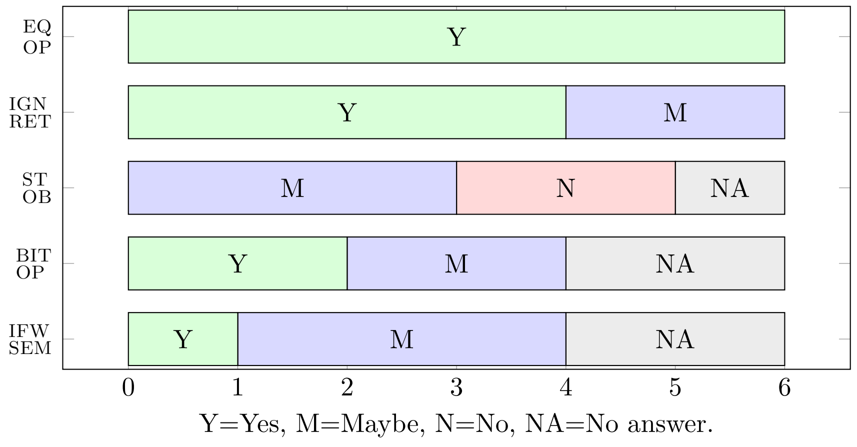


Figure 3.3: Focus group answers to Q2: "Is it likely that an INF101 student makes this mistake?"

Figure 3.4 shows the answers to the participants when asked if the errors would be more likely to make when being a beginners students in Java and knowing Python before. The group thinks $\genfrac{}{}{0pt}{}{\text{IFW}}{\text{SEM}}$ is the most likely error for this, and $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$ error is also a likely error. The $\genfrac{}{}{0pt}{}{\text{BIT}}{\text{OP}}$ and $\genfrac{}{}{0pt}{}{\text{ST}}{\text{OB}}$ errors are not likely to be more relevant when knowing Python before, with $\genfrac{}{}{0pt}{}{\text{IGN}}{\text{RET}}$ being the least likely. The participants who answered either `yes` or `maybe` on this answer for $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$ meant that the students would use the equal operator on object because they were used to it in Python. The participant who answered `no` said that the students would be confused by this regardless, because it is used to comparing primitive types in Java. Regarding $\genfrac{}{}{0pt}{}{\text{IFW}}{\text{SEM}}$, some participants thought it would be relevant because the syntax is familiar to Python. On the contrary, a participant argued that $\genfrac{}{}{0pt}{}{\text{IFW}}{\text{SEM}}$ would be less relevant, because students from Python would be more likely to underuse the semicolons than overuse them not having used them before.

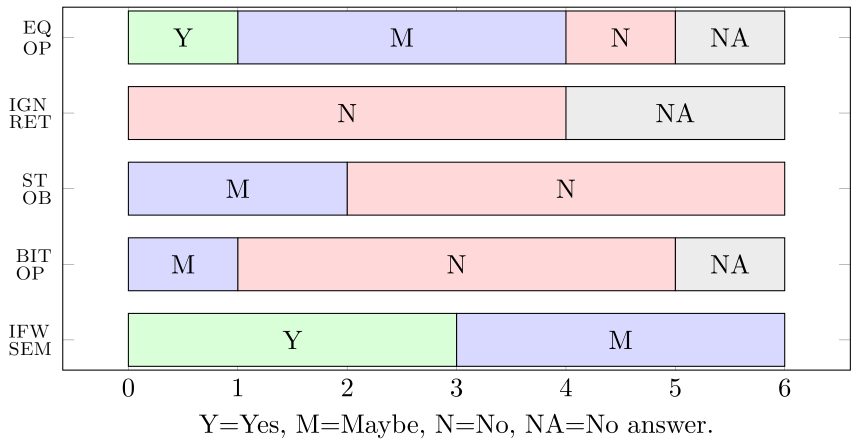


Figure 3.4: Focus group answers to Q3: "Would a student that knows Python beforehand be more likely to make this mistake than a student who does not know Python?"

The participants were asked to suggest other errors to get more input on what errors might be relevant. The group suggested these errors:

If No Brackets Error ($\genfrac{}{}{0pt}{}{\textbf{IFNO}}{\textbf{BRKT}}$). If you have an if-statement without brackets, the then-branch will only consist of the following statement. Even though the following statements are indented, Java only reads the first statement as the then-branch.

Field With No Initialiser ($\genfrac{}{}{0pt}{}{\textbf{FINO}}{\textbf{INIT}}$). If you use a field without initialising it, you get a `NullPointerException`. It was also commented that experienced programmers tend to make this mistake. $\genfrac{}{}{0pt}{}{\text{FINO}}{\text{INIT}}$ gives a `NullPointerException`, making it an Explicit Runtime Error (ERE), because it tells the programmer what is wrong. Therefore, even though it might be relevant, it does not pass the first criteria to be included in this thesis.

No Equals Method Error ($\genfrac{}{}{0pt}{}{\textbf{NOEQ}}{\textbf{METH}}$). It was mentioned that the students generally had problems with understanding the equals method and that this method is something they need to implement in Java.

Forget To Call Supermethod In Subclass ($\substack{\textbf{NOSUP}\\ \textbf{METH}}$). If you want to add functionality to a method in a subclass, you have to call the super method inside it. However, this violates the third criteria for including an error in this thesis by having to analyse several files.

Forget to annotate with @Override ($\substack{\textbf{NO}\\ \textbf{OVRD}}$). When overriding a method, for example the equals method, it should always be annotated with `@Override` to get a warning if you have misspelt the name of the method you are overriding. $\substack{\text{NO}\\ \text{OVRD}}$ also contradicts the third criteria, demanding other files to be analysed.

3.3.5 Discussion

The group agrees that $\substack{\text{EQ}\\ \text{OP}}$ is the most relevant error, while $\substack{\text{ST}\\ \text{OB}}$ is the least relevant error for the students. Then, the group believes the $\substack{\text{IGN}\\ \text{RET}}$ is very relevant to the students but not at all affected by knowing Python beforehand. The $\substack{\text{BIT}\\ \text{OP}}$ error might be relevant to the students but is not more likely to be made by a student that knows Python before. The $\substack{\text{IFW}\\ \text{SEM}}$ error was rare but might be more likely to make when knowing Python. At last, the $\substack{\text{ST}\\ \text{OB}}$ error was found to be less relevant in all cases.

As expected from the literature review, both $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{IGN}\\ \text{RET}}$ are indicated as relevant and frequent errors. The contradicting result of $\substack{\text{EQ}\\ \text{OP}}$ being more frequent than $\substack{\text{IGN}\\ \text{RET}}$ is expected due to considering a broader scope of the $\substack{\text{EQ}\\ \text{OP}}$ error than Brown and Altadmri [15]. Also, as expected, the group agrees that $\substack{\text{IGN}\\ \text{RET}}$ is not affected by learning Python before. The group did not think the $\substack{\text{ST}\\ \text{OB}}$ error was relevant nor affected by learning Python beforehand. $\substack{\text{ST}\\ \text{OB}}$ is omitted from Brown and Altadmri [15]'s study, so we can not compare the results, but given that the group agreed on the error, we feel more confident in the answers. A more complex error to analyse if it is relevant or not would be $\substack{\text{BIT}\\ \text{OP}}$. With only two in the group who had seen it before and the majority have not, the group did not give a clear indication as to the previous ones. However, the group seem to think that $\substack{\text{BIT}\\ \text{OP}}$ might be relevant but not caused by a transfer from Python.

An interesting result was the $\substack{\text{IFW}\\ \text{SEM}}$ error. Here the group think it is a rare error, yet they believe the students would likely transfer their semantics from Python to Java and be relevant in this sense. This response was a bit off given that the majority of the group have said they did not find the $\substack{\text{IFW}\\ \text{SEM}}$ relevant or have not seen it. Because this was the first error presented to the group, it might be that the participants misinterpreted the question giving a skewed result. The contradicting result of $\substack{\text{IFW}\\ \text{SEM}}$ being less relevant than $\substack{\text{BIT}\\ \text{OP}}$ is again expected because $\substack{\text{IFW}\\ \text{SEM}}$ has a smaller scope than in the study by Brown and Altadmri [15].

The group did not vote for the $\substack{\text{IFNO}\\ \text{BRKT}}$ because a participant suggested it, but Rosbach [16] mentions this as an ***Incorrect grouping*** error made by Java students. Additionally, $\substack{\text{IFNO}\\ \text{BRKT}}$ is the issue that caused the `goto fail;` bug mentioned in section 1.1. The group agreed that $\substack{\text{IFNO}\\ \text{BRKT}}$ would be relevant for the Python students because it is caused by not using brackets but indentation. $\substack{\text{NOEQ}\\ \text{METH}}$ was also suggested by a participant, and the lecturers agreed that $^{\text{NOEQ}}_{\text{METH}}$ is common among the students throughout the semester. The group agreed that students, in their experience, have difficulties understanding the importance of implementing the equals method in a class. Still, we do not have any previous data to compare this. The equals method is something the students are not used to from the previous course, so the $^{\text{NOEQ}}_{\text{METH}}$ error was included as well. Even though the errors $^{\text{BIT}}_{\text{OP}}$ and $^{\text{NOEQ}}_{\text{METH}}$ are not strictly related to a negative semantic transfer between Python and Java, they are included for further investigation in this thesis to see if they are relevant to the students.

3.4 Relevant errors for this study

Initially, we wanted to find relevant errors in order to investigate:

RQ1: *What semantic errors do students need help with when transferring from Python to Java?*

By doing a literature review and a focus group, we present the errors in Table 3.4, which is thought to be relevant for further investigating the question above. Using a mixed method in chapter 6 we hopefully will get closer to an answer to **RQ1**. For the rest of this thesis, by the phrase *these/the errors* we mean the errors found relevant in Table 3.4.

Error	Description	Relevant
EQ OP	Using == to compare objects	**Yes**
IGN RET	Ignoring return value from method call	No
ST OB	Call static method on object	No
INT DIV	Expect double from integer division	**Yes**
IFNO BRKT	If-statement without brackets	**Yes**
NOEQ METH	Not implementing equals method in class	**Yes**
BIT OP	Confusing bitwise operators with conditional operators	**Yes**
IFW SEM	Put a semicolon after if-condition	**Yes**
FINO INIT	Forget to initialise field	No
NOSUP METH	Not call supermethod	No
NO OVRD	No override annotation	No

Table 3.4: Collection of errors from literature review and focus group

Chapter 4

The detection of semantic errors and automating feedback

In this chapter, we go through the errors found to be relevant from Table 3.4 and aim to partly answer

---

RQ2: *How can we develop a tool to automate feedback for semantic errors when transferring from Python to Java?*

---

by looking at how existing environments handle the errors in section 4.1, discuss false positives and challenges by automating these errors in section 4.2, how to detect the errors using static analysis in section 4.4 and investigate how an error should be presented in section 4.5. A concrete description of the implementation of Uncoil can be found in chapter 5.

4.1 Existing error messages in other environments

We reconstructed the errors and registered the warnings generated by four different Integrated Development Environment (IDE)s: IntelliJ[1] Ultimate, Intellij Community, Visual Studio Code[2] and Eclipse[3]. By finding other instances where the errors are being automated, we can get an idea of how to create Uncoil and what might be lacking from existing tools to automate the errors.

[1]https://www.jetbrains.com/idea/
[2]https://code.visualstudio.com/
[3]https://www.eclipse.org/

The plugins used were downloaded for Mac OS with versions:

- IntelliJ IDEA 2021.3.1 (Ultimate Edition)
- IntelliJ IDEA 2022.1.1 (Community Edition)
- Visual Studio Code 1.67.2 (Universal)
- Eclipse IDE for Java Developers Version: 2021-12 (4.22.0)

We chose these IDE's because they are recommended in the INF101 course in the semester of writing this thesis. We can see in table Table 4.1 that only IntelliJ has warnings for the errors, and the two editions, Ultimate and Community, warn about different errors.

Error	IntelliJ	Eclipse	VSCode
$\substack{\text{EQ}\\ \text{OP}}$	New object is compared using '==' (Ultimate edition)	None	None
$\substack{\text{NOEQ}\\ \text{METH}}$	None	None	None
$\substack{\text{INT}\\ \text{DIV}}$	Integer division in a floating-point context (Community edition)	None	None
$\substack{\text{IFNO}\\ \text{BRKT}}$	Suspicious indentation (Community edition)	None	None
$\substack{\text{BIT}\\ \text{OP}}$	None	None	None
$\substack{\text{IFW}\\ \text{SEM}}$	If statement has empty body (Ultimate and Community edition)	None	None

Table 4.1: Different IDEs warnings.

Comparing the IntelliJ Community edition with the Ultimate edition, we see that Community warns for $\substack{\text{IFNO}\\ \text{BRKT}}$, $\substack{\text{IFW}\\ \text{SEM}}$ and $\substack{\text{INT}\\ \text{DIV}}$ while lacking a warning for $\substack{\text{EQ}\\ \text{OP}}$. In contrast, the Ultimate edition only warns for $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{IFW}\\ \text{SEM}}$. We believe a tool is needed to warn for all the errors, and as indicated from the literature review and focus group, we should automate feedback for $\substack{\text{EQ}\\ \text{OP}}$. It is expected that none of the environments warns for $\substack{\text{NOEQ}\\ \text{METH}}$, but we are somewhat surprised by not being warned of the $^{\text{BIT}}_{\text{OP}}$ error, being the cause of the ChromeOS bug mentioned in section 1.1. Because three of the errors are warned by the Intellij Community edition, it might be an idea to recommend this IDE for novices. However, then the students will not be warned of the crucial $^{\text{EQ}}_{\text{OP}}$ bug, so we believe that an additional tool to automate these error messages is needed. During the testing of the errors it was observed that IntelliJ uses highlighting in the code to present the errors.

We noticed some problems with the $^{\text{EQ}}_{\text{OP}}$ error when testing for this in Intellij Ultimate. The first was when objects where given as parameters to a method and compared using the equals operator, the warning was not generated. This probably is intentional, but in our case we want a warning for this as well. The second was that the warning was not flagged when comparing two ArrayLists or arrays using the equals operator, which is almost always wrong. To make troubles worse, as noted in section 2.2.3, the equals method behaves the same as the equals operator on arrays. We think this can further frustrate the student, being insecure of when the equals method is safe to use, so we aim to tailor the suggestion to this special case. Further, the warning for $^{\text{IFW}}_{\text{SEM}}$ only states that the if-statement is empty. While this is not untrue, we believe it can be frustrating for a student to look at the code and, in contrast to the message, observe a non-empty if-statement. Therefore, we aim to create the error message for $^{\text{IFW}}_{\text{SEM}}$ to explain to the student why Java reads this as an empty statement.

4.2 Should the feedback for these errors be automated?

In this section, we investigate if the errors should have automated feedback by discussing the target group, risk of false positives and how to ignore error messages.

4.2.1 Target group

Uncoil is targeted at novice Java students that know Python beforehand. Therefore we believe the errors would be less beneficial to automate for experienced developers. However, if the code containing the Chrome OS bug and `goto fail;` bug mentioned in section 1.1 were analysed for these errors, the bugs would probably not happen. Suppose we changed the error messages to target experienced Java developers and show all the error messages at once. In that case, we believe it would be helpful to give warnings for these errors during code reviews or before deploying code to production. As we will see in chapter 5, Uncoil can easily be adapted for further use, and other applications can tailor the results from the analysis to show multiple errors or change the presentation of the messages.

4.2.2 False positives and ignoring error messages

A false positive is when we do not have an error but still get an error message, and this is an important aspect to consider when deciding if these errors should be automated. False positives for these errors are not hard to find because they are based on the programmer's intent. Some errors may be more likely to have false positives than others. For example, the $\substack{\text{IFNO}\\\text{BRKT}}$ has a very clear case of when it is an error, like in Listing 2.6, so it gives few false positives. For the $\substack{\text{BIT}\\\text{OP}}$, there can be many false positives: often, two booleans evaluated by a bitwise operator is not a problem.

We see that the warning for IntelliJ in Table 4.1 warns about a ”suspicious indentation”, only flagging an error when there is second statement in an if-statement without brackets. Generally we want Listing 4.1 and Listing 4.2 to generate an error while Listing 4.3 should be ignored.

```
if (condition)
    m1();
    m2();
```

Listing 4.1: An invalid if-statement.

```
if (condition)
    m1(); m2();
```

Listing 4.2: An invalid if-statement.

```
if (condition)
    m1();
m2();
```

Listing 4.3: A valid if-statement.

Figure 4.1: Examples of valid and invalid if-statements

As mentioned, there might be many cases of $\substack{\text{BIT}\\\text{OP}}$ without it being a problem. This might be why there is no warning for this mistake by IntelliJ in Table 4.1, but being the cause of the Chrome OS bug mentioned in section 1.1, we will try to automate $\substack{\text{BIT}\\\text{OP}}$. We do not have any thoughts on how to limit the false positives for $\substack{\text{BIT}\\\text{OP}}$. The $\substack{\text{INT}\\\text{DIV}}$ is a false positive when we expect an integer. We see in Table 4.1 that IntelliJ has solved this by warning for this error in a ”floating-point context”. We have accomplished this in Uncoil, by ignoring the cases of Listing 4.6 and flagging an error for Listing 4.4 and Listing 4.5. We do this by going up the AST and check the expected type of either the method declaration or variable declaration. Other instances of false positives for $\substack{\text{INT}\\\text{DIV}}$ are out of the scope of this thesis.

```
double a = 7/5;
```

Listing 4.4: An invalid integer division

```
public double m() {
    return 7/5;
}
```

Listing 4.5: An invalid integer division

```
int a = 7/5;

public int m() {
    return 7/5;
}
```

Listing 4.6: Valid integer divisions

Figure 4.2: Examples of valid and invalid integer divisions

We also recognise that the $\substack{\text{NOEQ}\\\text{METH}}$ can give many false positives. A possible way to limit the false positives for $\substack{\text{NOEQ}\\\text{METH}}$ would be to search for usages of the equals method of the analysed class in the codebase. If it is used, we could warn the student that the equals method is used but not implemented. However, this is out of the scope of this thesis.

To combat *alert fatigue* caused by many false positives [42], we need a way to ignore error messages . Listing 4.7 shows an example where the errors $\substack{\text{NOEQ}\\\text{METH}}$ and $\substack{\text{INT}\\\text{DIV}}$ are ignored by Uncoil using annotations. We chose to use annotations to avoid being dependent on a specific user interface to ignore the errors, and to be able to configure the errors directly in the code.

```java
@NoEqualsMethod
@IntegerDivisionAllowed
class A {

   public double divide(int a, int b) {return a/b;}

}
```

Listing 4.7: $\substack{\textbf{INT}\\\textbf{DIV}}$ example Java with annotation

Results from [43] indicate that students use printing to debug code. For this reason, an error is not flagged if it is inside a print statement. We need to be able to compare two objects using the equals operator when implementing the equals method. Therefore the $\substack{\text{EQ}\\\text{OP}}$ is disabled when it is inside the equals method declaration.

4.3 Choosing analyses method

This section discusses which analysis method to use to discover the errors.

After the annotated AST is created by the semantic analyser, it can be sent to an optional code optimiser before the machine code (bytecode) generation [23]. Because of the optimisation, the bytecode loses some information, making it difficult to get back the source code [44].

One example of this is found in Listing 4.8, the $\substack{\text{IFNO}\\\text{BRKT}}$.

```java
public class DemoByteCode {

    public void method1() {}
    public void method2() {}

    public void m(boolean someCondition) {
        if (someCondition)
            method1();
            method2();
    }

}
```

Listing 4.8: Java if-statement without brackets

By compiling the file using a 16.0.1.hs-adpt compiler with the command

```
$ javac filename.java
```

we get the bytecode by using

```
$ javap -c filename
```

The bytecode for Listing 4.8 is in Listing 4.9

```
public class DemoByteCode {
  public DemoByteCode();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<
           init>":()V
       4: return

  public void method1();
    Code:
       0: return

  public void method2();
    Code:
       0: return

  public void m(boolean);
    Code:
       0: iload_1
       1: ifeq          8
       4: aload_0
       5: invokevirtual #7                  // Method method1:()V
       8: aload_0
       9: invokevirtual #12                 // Method method2:()V
      12: return
}
```

Listing 4.9: Java bytecode for Listing 4.8

According to the documentation [30, §6.5], if the if-condition is `true`, it will proceed to line 4 in `public void m(boolean)`, the line under the if-condition. If the condition is `false`, it will proceed to line number 8 in `public void m(boolean)`, which was meant to be inside the if-body. However, we can not tell if the programmer meant for line 9 to be inside the if-statement or not with this information.

If we were to use the annotated AST from the semantic analyser, we would get a tree as in Figure 4.4. The first thing we can see is that the `ThenStatement` does not have a `BlockStatement`. Furthermore, as we learnt in subsection 2.3.3, the annotated AST holds the semantics of the nodes as attributes. This includes the column position of the `ExpressionStatement`, and we can check if it has the same indention level as the `ThenStatement`.

The above example shows that AST analysis is best for these types of errors.

4.4 Detecting errors using the abstract syntax tree

This section describes an abstract approach on how to find the errors, while a concrete implementation is described in chapter 5.

4.4.1 Method

By creating an example for each error, we empirically investigate how to analyse code for the errors using semantic analysis on the AST. The AST have been simplified for brevity. The AST for each error has been created with the help of *Eclipse ASTView*[4], a plugin for visualising AST for a Java file in Eclipse.

4.4.2 Results

Equals Operator Error ($^{\mathbf{EQ}}_{\mathbf{OP}}$)

This error should be present when two objects are compared using the equals operator. An example of this can be viewed in Listing 4.10.

```java
Object a1 = new Object();
Object a2 = new Object();
boolean condition = a1 == a2;
```

Listing 4.10: Example of equals operator on object

The expression `a1 == a2` is an *infix expression*, because it uses an operator (`==`) and applies it to the left operand `a1` and the right operand `a2`. By using the visitor pattern, as explained in subsection 2.3.3, we can match on an infix expression in the source code. The AST for the expression `a1 == a2` is showed in Figure 4.3.

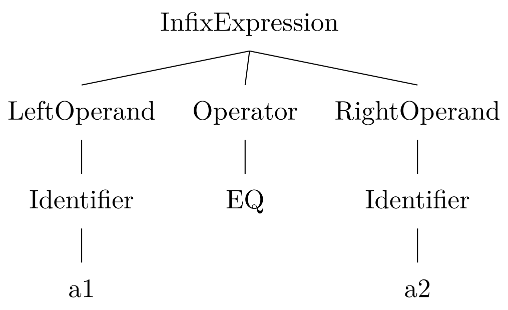


Figure 4.3: AST of $^{\mathbf{EQ}}_{\mathbf{OP}}$

[4]https://www.eclipse.org/jdt/ui/astview/index.php

The first thing to check is that the operator is the EQ sign (`==` or `!=`). Since we need to be able to use the equal operator on primitive types, we check that the operands are objects by looking up the types in the symbol table. In addition to checking that the operands are not primitives, we also need to check that the operands are not `null`. The latter ensures that we can perform a null-check on an object. By traversing the tree upwards, we can find the ancestor to the `InfixExpression`. We need to check if one of the ancestors is a print statement or equals method declaration to limit the false positives as described in subsection 4.2.2. For the special case for arrays as explained in section 2.2.3, we check if the type for the operands are arrays allowing us to create a different suggestion for this specific error.

If No Brackets Error ($\binom{\textbf{IFNO}}{\textbf{BRKT}}$)

This error occurs when the if-statement has no brackets, and the statements after the condition are indented in such a way that they look like they should belong to the if-body, illustrated in Listing 4.11.

```
boolean condition = false;
if (condition)
    method1();
    method2();
```

Listing 4.11: Example of if no brackets error

As explained thoroughly in subsection 2.2.3, the statement `method2();` will be executed regardless of the if-statement. By using the visitor pattern we can match an if-statement, and analyse it. We construct the AST for the if-statement in Figure 4.4 for the example in Listing 4.11.

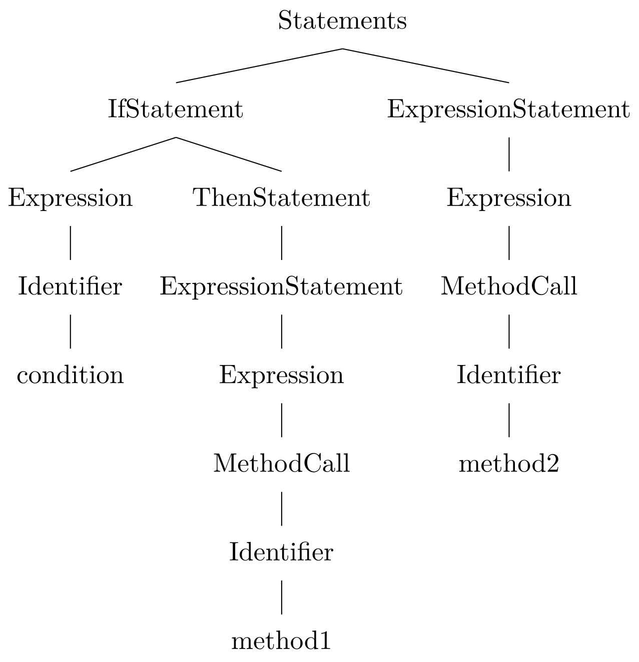


Figure 4.4: AST of $\substack{\textbf{IFNO}\\\textbf{BRKT}}$

First, we need to check that the if-statement is without brackets. This is easily done by checking the type of the child to the `ThenStatement`. In Figure 4.4 we see that the type is an `ExpressionStatement`, confirming that this if-statement is without brackets. If the if-statement has no sibling, we have no error because there is nothing following the body of the if-statement. To find out if we have a sibling or not, we go up to the parent and check how many children it has. In Figure 4.4 the rightmost `ExpressionStatement` is the sibling, so we need to check if this `ExpressionStatement` has the same indentation level as the body of the if-statement. We also need to check if it is on the same line as the body. As discussed in subsection 2.3.3 the nodes in the AST are decorated with the semantics of the node attached to it. This often includes the node's position, so we can extract the indentation level and line number of the sibling to check for this error. In general, we want Listing 4.1 and Listing 4.2 to produce an error, while Listing 4.3 is valid as discussed in subsection 4.2.2.

Integer Division Error ($\binom{\textbf{INT}}{\textbf{DIV}}$)

This error appears when we do an integer division in Java and expect a decimal number in return. Section 2.2.3 goes through the difference between integer division in Python and Java. Even though we are casting to a `double` in Listing 4.12 we will get `1.0` as an answer. The AST of the expression `7/5` is illustrated in Figure 4.5.

```java
double a = 7/5;
```

Listing 4.12: Example of integer division in Java

The expression `7/5` is also an infix expression like $\substack{\text{EQ}\\ \text{OP}}$. However, now the operator is the dividing operator (DIV), and the operands are integers.

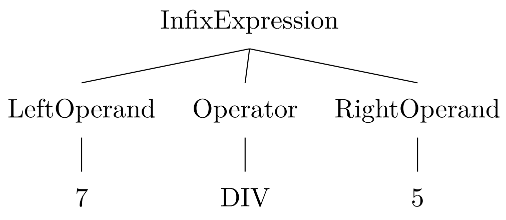


Figure 4.5: AST of $\substack{\textbf{INT}\\ \textbf{DIV}}$

The approach is very similar to detecting the $\substack{\text{EQ}\\ \text{OP}}$: When visiting an `InfixExpression` in the AST, we then need to check that the operator used is the dividing operator, and if the operands are integers by using a symbol table. Also, we need to check if one of the ancestors is a print statement. We get a false positive of this error by expecting an integer from the division, as discussed in subsection 4.2.2. Therefore, we need to be able to have statements like `int a = 7/5;` without producing an error. We can achieve this by walking up the tree, matching on a `VariableDeclaration` or `MethodDeclarations` and checking its type. If it is of type `int`, we can discard the error.

Semicolon After If Error ($\substack{\textbf{IFW}\\ \textbf{SEM}}$)

This error is caused by a semicolon after the condition in the if-statement. Listing 4.13 shows an example of this, and Figure 4.6 shows the AST of the error.

```java
if (condition) ; {method1();}
```

Listing 4.13: Example of semicolon after if in Java

The AST of the statements are illustrated in Figure 4.5.

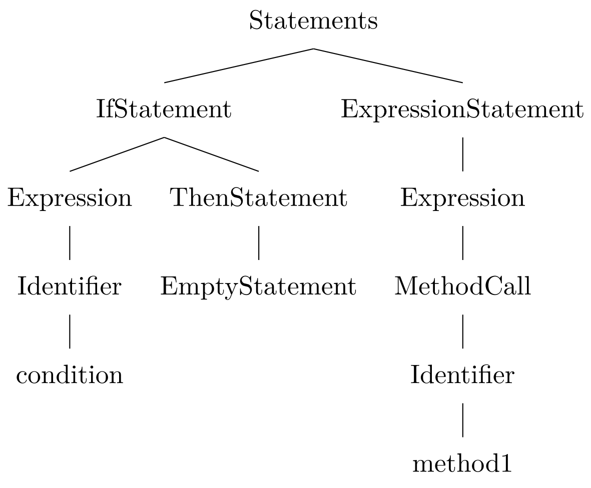


Figure 4.6: AST of $\substack{\textbf{IFW}\\ \textbf{SEM}}$

We can find this error in the AST by visiting if-statements, and check if it has an empty `ThenStatemement`. This differs from an empty block, where the `ThenStatement` would be a `BlockStatement`.

Bitwise Operator Error ($\substack{\textbf{BIT}\\ \textbf{OP}}$)

This error occurs when evaluating boolean expressions with one of the bitwise operators illustrated in Listing 4.14.

```java
boolean condition = true & false;
```

Listing 4.14: Example of bitwise operator in Java

This is also an infix expression like $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{INT}\\ \text{DIV}}$. The AST of the boolean expression `true & false` is drawn in Figure 4.7.

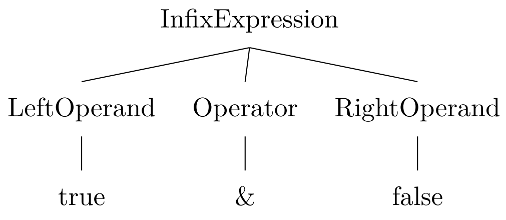


Figure 4.7: AST of $\substack{\textbf{BIT}\\ \textbf{OP}}$

The approach to detect this error is very similar to $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{INT}\\ \text{DIV}}$. If the left operand and right operand are of type `boolean`, we need to check if the bitwise operators have been used. Additionally, we need to check if one of the ancestors is a print statement.

No Equals Method Error $\binom{\textbf{NOEQ}}{\textbf{METH}}$

This error should be flagged when a class without an equals method is implemented.

```java
class A { public void m() {} }
```

Listing 4.15: Example of missing equals method in Java

The AST of the class is illustrated in Figure 4.8.

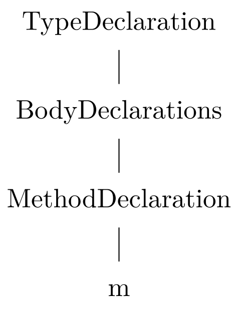


Figure 4.8: AST of $\substack{\textbf{NOEQ}\\\textbf{METH}}$

By visiting a class declaration in the AST, we can get the body of the class. The body holds the field declarations and method declarations. To find this error, we need to go through the list of declarations in the body and find all `MethodDeclarations`. If we find a `MethodDeclaration`, we need to check that the name is `equals` and the return type is `boolean`. If the body has no `MethodDeclarations`, or no equals method, we flag an error. The error will not be flagged if it is an interface or abstract class.

4.5 The structure of the error messages

This section lists how previous studies have investigated how to present error messages to students. By building on previous knowledge, we contribute to answering how we can create the error messages for Uncoil and, in turn, partly answer RQ2.

Studying what students found helpful in error messages, Becker [45] found that explaining the error in simple terms and providing a solution is helpful for students. Furthermore, Brækken [46] found in her studies that when providing an example for the students to describe an error, it may confuse the students: Some of the students in her research thought that the example was the solution. The structure of the message is essential for students to understand the error message according to [47]. Additionally, a recent study [48] has tried to find out what helps students understand error messages. The key factors they found were message length, jargon and sentence structure. Similar tools [20] [22] to Uncoil have displayed the different code side by side, but we focused on creating a suggestion.

Considering previous research, the error messages should be on point, written in simple terms, and have suggestions. A link is added with a more comprehensive explanation for each error to avoid long error messages. A suggestion is displayed with the error message tailored to the analysed code to avoid confusion. For example, if we analyse the code in Listing 4.16

```java
@NoEqualsMethod
class A {

   public double divide(int a, int b) {return a/b;}

}
```

Listing 4.16: $\substack{\textbf{INT}\\\textbf{DIV}}$ example Java

the error message is shown in Listing 4.17.

```
1  In class A, on line number 4
2
3  You are doing an integer division! In Python, you could divide two
4  integers and get a decimal as a result. In Java we need to change the
5  integers to decimals before we divide to get the same result.
6
7  You should try (double)a/(double)b
8
9  More info? Check out:
10
11 https://master-thesis-frontend-prod.herokuapp.com/integerdivision
```

Listing 4.17: Error message produces by $\substack{\textbf{INT}\\\textbf{DIV}}$

Note that we have used the word ”decimal” instead of ”double” to use simple terms. On line one in Listing 4.17, we see the first line of the error message. This should tell where to find the error. On line number three, the first sentence describes what the mistake is. The preceding sentences describe why this is a problem and how it relates to Python. The seventh line proposes the suggestion for how to fix the error and is adapted to the source code. The last section of the error message holds a link with more information, where the student can get more into the depth of the problem. The jargon is kept on a non-formal level. It has been chosen to show only the first error not to overwhelm the students in thread with other similar tools [22].

Generally, the error message have the form in Listing 4.18:

```
[POSITION]

[CAUSE OF THE ERROR] [EXPLANATION]

[SUGGESTION]

[MORE INFO]
```

Listing 4.18: Error message form

4.6 Discussion

To get closer to an answer to RQ2, we have constructed the AST for each error and described a method to find the errors in the source code. We have proposed some methods for how to avoid false positives for $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$, $\genfrac{}{}{0pt}{}{\text{INT}}{\text{DIV}}$ and $\genfrac{}{}{0pt}{}{\text{IFNO}}{\text{BRKT}}$ and, even though not in the scope of this thesis, sketched a solution for how to avoid false positives for $\genfrac{}{}{0pt}{}{\text{NOEQ}}{\text{METH}}$ as well. Furthermore, we propose to use annotations to ignore error messages. We have chosen to represent the errors in a text format using natural language. Based on previous research, we have found that an error message should contain the position, cause of the error, a suggestion and a link with more info. We have discussed the target groups for Uncoil and specified that we believe the error messages should be automated for beginners in Java that knows Python. However, Uncoil could be further developed to target more experienced Java developers. In section 4.1 we found some limitations to existing tools, lacking functionality we believe is essential for a novice Java student. We proposed a solution for developing the features needed and believe that a tool like Uncoil is needed to automate the errors.

As noted in section 4.1 IntelliJ highlights the errors in the code to present them. However, Uncoil could not implement this due to time constraints and limited access to the code being analysed. We will go more into detail about this in chapter 5. We saw in section 4.3 that we could not use bytecode analysis to find these errors. Consequently, we could not have used SpotBugs as an analysis tool to find them. SonarQube[5] is another tool that was considered to find the errors but was discarded. We considered it too heavy-weight for the relatively small scope of analysis. Additionally, we could make the error messages use more jargon as the students gets more advanced, possibly like a language localisation tool.

[5]https://www.sonarqube.org/

Chapter 5

Design, implementation and development of Uncoil

This chapter discusses the implementation of the concepts in chapter 4 and the challenges with implementing it. Specifically, we answer

---

RQ2: *How can we develop a tool to automate feedback for semantic errors when transferring from Python to Java?*

---

by implementing a solution. The first section 5.1 shows a demonstration of Uncoil. Then we go over the overall architecture of how Uncoil is built in section 5.2. Further, in subsection 5.2.2, subsection 5.2.3 and subsection 5.2.4 we go deeper into the implementation and usages of the different components of Uncoil. In section 5.3 we go trough the development methods and present a use case for Uncoil to uncover the functionalities needed. In subsection 5.3.3 we describe how Uncoil is tested during development and deployment. At last, in section 5.4 we discuss how Uncoil should be distributed and provide an example of how Uncoil can be used for further work.

5.1 Demonstration

The code in Listing 5.1 is an example of an $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$. By using Uncoil on this code we get the error message in Listing 5.3.

```java
@NoEqualsMethod
class A {
    public void method(A a1, A a2) {
        if (a1 == a2) {System.out.println("a1 is equal to a2");}
    }
}
```

Listing 5.1: Simple example of $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$

```
In class A, on line number 4

You are using "==" to compare objects! In Python you could do this,
    but in Java we use the equals method.

You should try a1.equals(a2)

More info? Check out:

https://master-thesis-frontend-prod.herokuapp.com/equalsoperator
```

Listing 5.2: Feedback from Uncoil in this thesis detecting $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$

5.2 Code structure

5.2.1 Architecture

The *architecture* of an application shows how it is structured and set together [49], and the overall architecture for Uncoil is shown in figure Figure 5.1. *Maven*[1] is a build tool that handles project dependencies, so we do not have to download them manually, and a *Maven repository*[2] holds artefacts and dependencies. *Uncoil core* is a Maven package, `master-thesis-backend-analyser`, that performs the analysis of the given source code. The `master-thesis-annotations` holds the annotations that can be used to ignore the errors. Both `master-thesis-backend-analyser` and `master-thesis-annotations` are accessible from a Maven repository deployed on *repsy.io*[3], where you can administrate and deploy your repositories. Maven was used because it is well-known among developers, and the dependency is usable by other applications. Other applications can either download the package as a dependency in their application or use the Application

[1]https://maven.apache.org/
[2]https://maven.apache.org/guides/introduction/introduction-to-repositories.html
[3]https://repsy.io/

Programming Interface (API): `master-thesis-backend-api`. An API lets instances developed with different programming languages communicate with each other, enabling other applications to use Uncoil.

The architecture for Uncoil is *layered* [49], separating different concerns: presenting the data using `master-thesis-web-frontend`, control the action of a request using `master-thesis-backend-api`, and analyse the code using `master-thesis-backend-analyser`. The layers are named, respectively, the representation, controller, and business layers. Uncoil does not have a data layer. The `master-thesis-web-frontend` and `master-thesis-backend-api` are deployed in two different instances on a Platform as a Service (PaaS), providing the infrastructure needed to run and scale the instances [50]. As a PaaS we chose to use Heroku[4] to deploy Uncoil, and by being available for users through the browser, Uncoil is a Software as a Service (SaaS) [50]. The layers are deployed independently, taking the web-application in the direction of a *microservice architecture* [49]. Furthermore, the presentation layer and controller communicate with each other using a remote access API that speaks in favour of being a microservice architecture. The `master-thesis-backend-analyser` is also deployed separately but does not allow remote access - the package needs to be downloaded by `master-thesis-backend-api` and deployed as a *monolithic application* [49] on Heroku. Nevertheless, other applications can use `master-thesis-backend-analyser` by downloading it as a dependency to enable future work.

[4] https://www.heroku.com/home

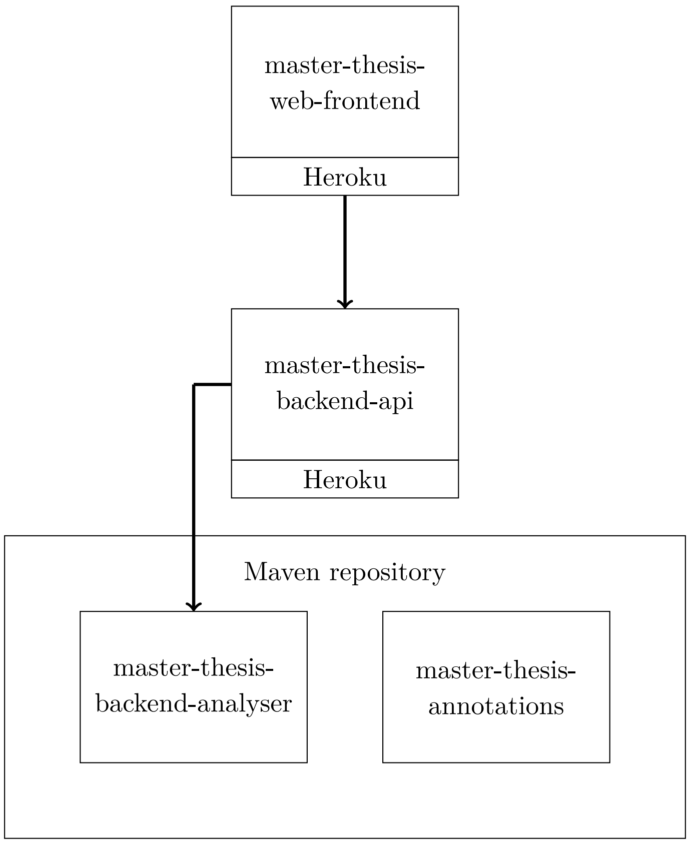

master-thesis-web-frontend
Heroku
master-thesis-backend-api
Heroku
Maven repository
master-thesis-backend-analyser
master-thesis-annotations

Figure 5.1: Architecture

5.2.2 Implementation of the analyser

The `master-thesis-backend-analyser` uses *`JavaParser`* [40], a tool to parse and analyse Java code. Other language handling tools were also considered, like ANTLR, but JavaParser was chosen due to the ease of setup with maven and the smaller scope of focusing on parsing Java.

Figure 5.2 is an UML (Unified Modelling Language) diagram [51] for the `master-thesis-backend-analyser`, describing how the analyser is structured. The diagram is somewhat simplified, but shows the main lines of the analyser. The heart of the `master-thesis-backend-analyser` package is the `BugFinderVisitor` class, that does the analysis on the source code and creates the `BugReport`. The `BugFinderVisitor` inherits from `VoidVisitorAdapter` from JavaParser, that visits the children of the nodes in the AST in random order [40]. Two classes inherits from `VoidVisitorAdapter`: `BugFinderVisitor` and `AnnotationsVisitor`. In Figure 5.1 `VoidVisitorAdapter` contains only the methods used in this thesis for brevity. The `BugFinderVisitor` visits the nodes in the AST created by JavaParser, and add errors to the `BugReport` if it is found. All errors inherit from `BaseError`, an abstract class implementing the common functionality across the errors, and provides a contract if the implementation needs to be done for each error. The abstract methods are marked with recursive text in Figure 5.2. The `BaseError` implementation allows for reuse of code, along with the implementation of `BinaryExprError` and `IfStatementError`. As we know from section 4.4 the errors $^{\text{EQ}}_{\text{OP}}$, $^{\text{INT}}_{\text{DIV}}$ and $^{\text{BIT}}_{\text{OP}}$ are all `InfixExpression`, and `IfWithoutBracketsError` and `SemiColonAfterIfError` are similar to each other. This implementation takes advantage of this, creating a common class `BinaryExprError` that implements their common functionality, avoiding duplicating code. We also do the same for `IfWithoutBracketsError` and `SemiColonAfterIfError` by letting both inherit from `IfStatementError`. From section 2.2.3 we have found one specific case of the `EqualsOperatorError`, where we should produce a different suggestion for the Arrays class: `Arrays.equals(array1, array2)`. This is solved by a flag in the `EqualsOperatorError` when comparing two arrays with the equals operator.

The `Analyser` class depends on the interface `AnalyserConfiguration`, where a concrete implementation of the interface can decide how errors should be ignored. The `AnnotationsAdapter` has the responsibility of transforming the annotations in the source code to suit the demands of the `AnalyserConfiguration`. Here we implement the *adapter pattern* [39], converting from annotations to the name of the error. The `AnnotationsAdapter` uses the `AnnotationsVisitor` to find the annotations in the source code. The annotations are coupled with the class it belongs to, allowing the configuration of errors to be class-based. The default configuration uses annotations in the source code, but it can easily be extended to use a user interface implementing the interface.

The `Analyser`'s method `analyse(String code)` takes in Java source code as text, and analyses it. If JavaParser gives an error during analysis, the error is propagated and stored in the `BugReport`. We can get the exception by calling `getException()`, that optionally gives an exception.

Uncoil can be adopted to different target groups - for example to experienced Java developers. One way to change the error messages could be to change the actual strings for each error, but a possible better way would be to implement the *strategy pattern* [39] in Java, allowing different strategies to create different instances of the errors. This way it could easily be shifted between novices in Java or experienced developers by using different strategies, and it could be added a new strategy for other groups. Also, a new application can match on the different types of errors and display the errors as wanted. However, this is left for further work.

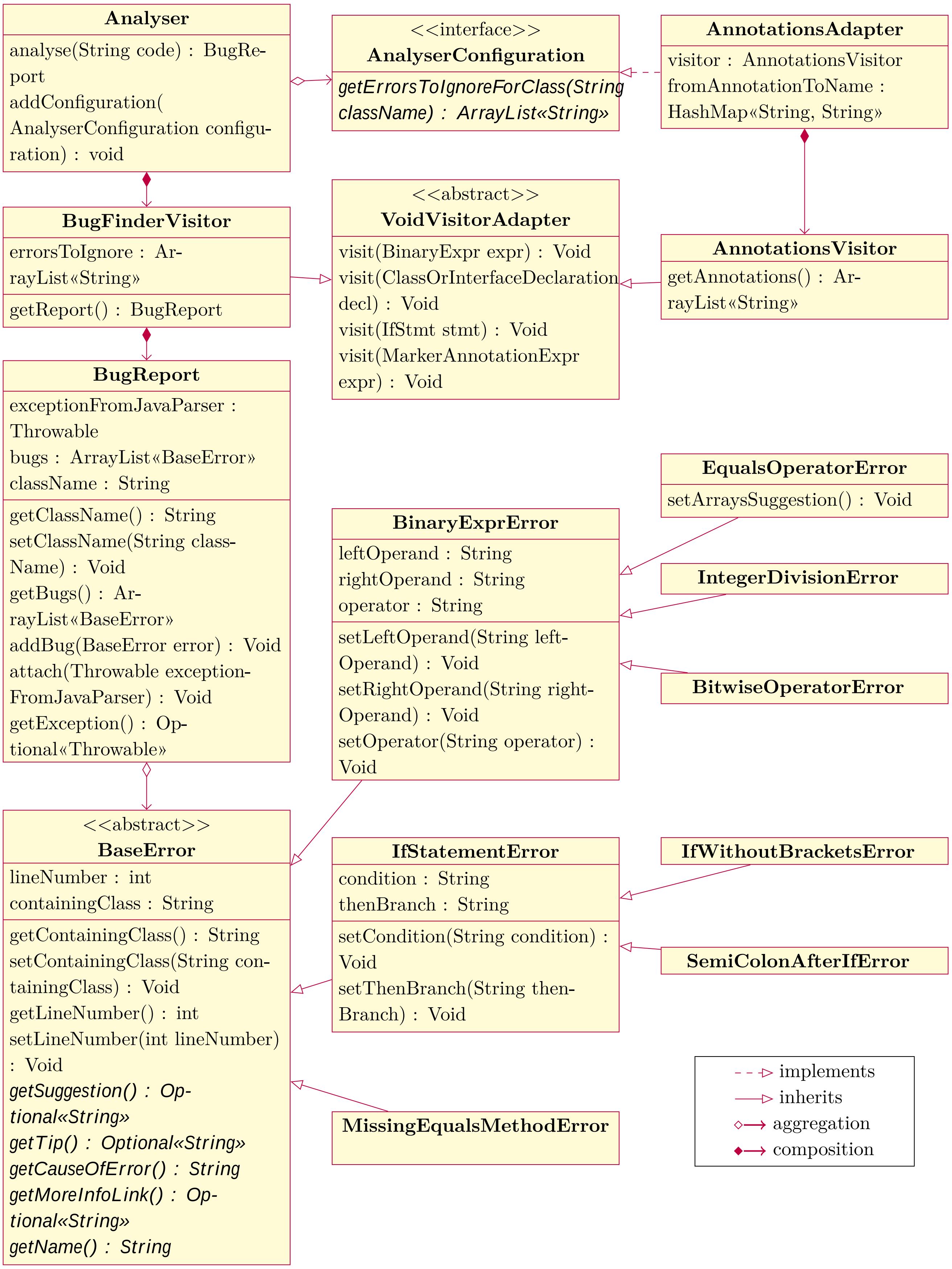

Analyser
analyse(String code) : BugReport
addConfiguration(AnalyserConfiguration configuration) : void
<<interface>>
AnalyserConfiguration
getErrorsToIgnoreForClass(String className) : ArrayList«String»
AnnotationsAdapter
visitor : AnnotationsVisitor
fromAnnotationToName : HashMap«String, String»
BugFinderVisitor
errorsToIgnore : ArrayList«String»
getReport() : BugReport
<<abstract>>
VoidVisitorAdapter
visit(BinaryExpr expr) : Void
visit(ClassOrInterfaceDeclaration decl) : Void
visit(IfStmt stmt) : Void
visit(MarkerAnnotationExpr expr) : Void
AnnotationsVisitor
getAnnotations() : ArrayList«String»
BugReport
exceptionFromJavaParser : Throwable
bugs : ArrayList«BaseError»
className : String
getClassName() : String
setClassName(String className) : Void
getBugs() : ArrayList«BaseError»
addBug(BaseError error) : Void
attach(Throwable exceptionFromJavaParser) : Void
getException() : Optional«Throwable»
EqualsOperatorError
setArraysSuggestion() : Void
BinaryExprError
leftOperand : String
rightOperand : String
operator : String
setLeftOperand(String leftOperand) : Void
setRightOperand(String rightOperand) : Void
setOperator(String operator) : Void
IntegerDivisionError
BitwiseOperatorError
<<abstract>>
BaseError
lineNumber : int
containingClass : String
getContainingClass() : String
setContainingClass(String containingClass) : Void
getLineNumber() : int
setLineNumber(int lineNumber) : Void
getSuggestion() : Optional«String»
getTip() : Optional«String»
getCauseOfError() : String
getMoreInfoLink() : Optional«String»
getName() : String
IfStatementError
condition : String
thenBranch : String
setCondition(String condition) : Void
setThenBranch(String thenBranch) : Void
IfWithoutBracketsError
SemiColonAfterIfError
MissingEqualsMethodError
implements
inherits
aggregation
composition

Figure 5.2: UML diagram of the analyser

5.2.3 Implementation of the API

The `master-thesis-backend-api` was created with *Spark*[5], a lightweight framework for creating endpoints in a web application. The Spring framework was also considered but discarded in favour of the easy set-up for Spark. The API is built with Maven and uses the analyser maven package when deployed. The API has two main endpoints: `/runjava` and `/analyse`. The first endpoint is created for evaluation purposes. It takes in Java code, stores the code in a temporary file on the server, executes it and returns the result. This lets participants of the study run Java code from the browser. The functionality for executing the Java code should be separated in another deployment to maintain a microservice architecture and for security reasons, but this was not done due to timing constraints. The second endpoint accepts a payload as Java code, analyses the code and returns the result in JSON such that the representation layer can parse it an represent it. An example of the JSON response from the API can be found at Figure 5.4.

According to [50] *RESTful (REpresentational State Transfer) API* is a *stateless* API that also follows conventions on how to handle requests: `GET` (get an instance), `POST` (create a new instance), `PUT` (update an instance) and `DELETE` (delete an existing instance). Having a stateless API means that two consecutive calls are independent from each other, and the service does not remember the previous call. Our API is a RESTful API, following the needed conventions. Consequently, the data that should be analysed needs to be sent and returned to the client in one request. For this purpose, the `HTTP POST` request is suitable [52], where the code to be analysed is given as a payload in the request, and the server can return the result. We use the `POST` method instead of `GET` method due to security and length restrictions [52]. When sending a `GET` request, the data is visible in the URL and with a maximum of 2048 characters we would not be able to analyse large Java files [52]. The `POST` method is somewhat more secure in that the data is stored in the payload of the request, and it does not have a max length [52].

By sending the data in Figure 5.3 as a POST request to the API, we get the response in Figure 5.4. `master-thesis-backend-api` creates the JSON and sends it to the client. The client side parses the response and presents the data in the browser. If the analyser returns an exception from the analysis, it is added to the JSON response.

```java
@NoEqualsMethod
class A {

    boolean a = true & false;

}
```

Figure 5.3: Example payload to `master-thesis-backend-api`

[5] https://sparkjava.com/

```
{
    "errors": [
        {
            "containingClass": "A",
            "suggestion": "replacing & with &&: true &&
               false",
            "type": "master.thesis.backend.errors.
               BitwiseOperatorError",
            "explanation": "You are using the bitwise
               operator (&). In Python we use "and" and "or"
                as boolean operators, but in Java we use "&&
               " (and) and "||" (or)!",
            "lineNumber": 4,
            "moreInfoLink": "https://master-thesis-frontend-
               prod.herokuapp.com/bitwiseoperator"
        }
    ]
}
```

Figure 5.4: Example JSON response from `master-thesis-backend-api`

5.2.4 Implementation of the user interface

The `master-thesis-web-frontend` is implemented using *JavaScript*[6] and is a lightweight web application. Uncoil consists of a simple editor, where you can paste in the code you want to analyse and use a button to send it for analysis at the backend. When the students evaluated Uncoil, they got access to a simple online editor with the tasks ready in the editor integrated with Uncoil. The students could run the code and check if it had passed the task or get a hint (use Uncoil).

Several attempts were made to highlight the code to make the code more readable in the editor. A simple tokeniser was made, where the keywords and identifiers to be highlighted were specified. Unfortunately, the tags that coloured the words made the editor unusable. It was also made an attempt to use embedded editors, *highlight.js*[7] and *CodeMirror*[8], to highlight the code. This worked locally but caused a dependency problem when deployed to a different environment. Due to little time, the highlighting of code was not prioritised. The editor has line numbers to localise the error. The editor has an observer that notifies about change in the editor, allowing the line numbers to be repopulated. A console below the editor to display the error was added after feedback from the first student. The tasks are created as a single page application, where the

[6] https://www.javascript.com/
[7] https://highlightjs.org/
[8] https://codemirror.net/

button for the previous task and next task updates the text in the editor. The error message provides a link to more info, which will open in a new tab. During evaluation of Uncoil we filtered the errors after retrieving them from the API instead of using annotations, to avoid students being confused by annotations in the code.

5.3 Development method

This section goes through the development method and presents a case to uncover the requirements for Uncoil. Further, we describe the methods used to Uncoil and the deployment methods for the artifacts.

5.3.1 Use case

A *usecase* describes the flow and requirements for a system, using *actors* and *goals*. The actor is either a person that uses the system or another system, and the goal is what the user wants to accomplish with the system. [53]

Use case: Analysing Java code with $\frac{\text{INT}}{\text{DIV}}$ error

Actor: The actor, in this case, will be an INF101 student, that knows Python before.

Context: It is the second week of learning Java, and the student is trying to implement a simple calculator. One of the methods should take two integers, divide them and return the result. The student ends up with the code in Listing 5.3.1.

```java
class Calculator {

    public static void main(String[] arg) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.divide(7,5));
    }

    public double divide(int a, int b) {
        return a/b;
    }
}
```

Testing the calculator with `a=7` and `b=5` gives the student a surprising result, namely `1.0`. Finally getting the code to compile, the student is frustrated to find out that it does not work as expected, knowing from Python that an integer division returns a decimal.

Having defined the context, we can now define the goal:

Goal: Find out why the output is wrong.

The student copy and paste the code into Uncoil, and press the button to analyse the code. Uncoil should behave as follows:

1. The code is retrieved from the text field and it is created a POST request with the code as the body.
2. The API retrieves the POST request and gets the Java code as a string from the request body.
3. An instance of the analyser is created and the API calls the analyse method with the source code.
4. The analyser parses the code and returns a parse error immediately if it is unsuccessful.
5. The analyser checks if annotations are present in the code and register them.
6. The analyser analyses the code for errors and returns them in a bug report.
7. The API creates a JSON containing a list of the errors, or if present an exception, from the report and sends it as a response.
8. The frontend parses the JSON and displays the response to the user, as seen in Listing 5.3.

```
In class Calculator, on line number 9

You are doing an integer division! In Python, you could divide two
integers and get a decimal as a result. In Java we need to change the
integers to decimals before we divide to get the same result.

You should try (double)a/(double)b

More info? Check out:

https://master-thesis-frontend-prod.herokuapp.com/integerdivision
```

Listing 5.3: Feedback from Uncoil in this thesis

5.3.2 Creating a Minimal Viable Product

A Minimal Viable Product (MVP) is a product where you serve the bare minimum of the requirements, and a product can have both functional and non-functional requirements. Functional requirements are those that are accessible to the user, and what the user could do with the software, while non-functional requirements are properties the application should have. [50]

The use case in subsection 5.3.1 uncovered many of the requirements, for example, that a parse error in the code should return an error. For the MVP we set the functional requirements as in Table 5.1 and the non-functional requirements as in Table 5.2.

Functional requirement	**Priority**	**Implemented**
Return an error message for the errors in this thesis	Need to have	Yes
Return error when code can not be analysed	Need to have	Yes
An error message should contain where it is and what causes the error	Need to have	Yes
Can ignore error messages	Nice to have	Yes

Table 5.1: Functional requirements

Non-functional requirement	**Priority**	**Implemented**
Easily accessible for the students	Need to have	Yes
Distributed in a known environment for the students	Nice to have	No
Separate analyser for easily maintaining	Nice to have	Yes
Scalability	Need to have	Yes

Table 5.2: Non-functional requirements

By using annotations, we have fulfilled the demand *Can ignore error messages*. We gave this requirement a lower priority because it is not strictly needed to analyse the code. Still, because of the risk of false positives, we wanted to include this in the MVP. We made Uncoil *Easily accessible for the students* by providing it as a SaaS, allowing for using Uncoil in the browser. The *Scalability* requirement is solved by deploying Uncoil on a PaaS. By *Distributed in a known environment to the students* we mean, for example, to give the feedback in their IDE. However, we did not manage to do this.

Development style

Agile methods are strategies for developing a product. We will not go into the depths of the different agile methods in this thesis but briefly explain that they favour a change of requirements during development over rigour requirements that can not change [54].

We have not adopted a specific agile approach, but Uncoil has been adapted to new requirements during development. For example, two plugins were developed with the desired functionality to meet the requirement *Distributed in a known environment for the students* from Table 5.2. Still, the plugins did not fulfil this requirement because they were created for different environments than the students used that semester. Deciding to construct a web application instead, we now had to add *Scalability* as a new requirement.

To make sure that the MVP meets the requirements in Table 5.1, we can test the application. As a part of agile methodology, to shorten the feedback loop and finding bugs early, we have used Test Driven Development (TDD) [50] to develop Uncoil. We have focused on *unit tests*, testing parts of the applications, and some *integration tests*, testing the application as a whole. Figure 5.5 and Figure 5.6 displays where in the development cycle the requirements are tested. Technical debt is when you postpone the refactoring of the code [55]. When doing TDD, you should ideally refactor once the tests pass, but due to time pressure, this did not always happen. It was willingly taken on some technical dept during development when creating the MVP before getting feedback from the user testing. However, after the evaluation of Uncoil the code has been refactored and made more understandable.

5.3.3 Deployment

Development cycle and feedback loop

When deploying software, we can set up a *pipeline* that consists of different *environments* for the application. We have used three environments in our development cycle: *Local environment*: where the development happens and local testing, *Development environment*: the application is deployed in another environment and tested, *Production environment*: the application is deployed to be accessible to the public. [56]

Figure 5.5 illustrates how the backend analyser is deployed to a Maven repository. The different environments are separated by a vertical line, and arrows indicate the flow between the different environments. The code is stored on the development platform *GitHub*[9]. When pushed to GitHub, the unit tests are run automatically using *GitHub Actions*[10]. In this case, GitHub Actions act as a development environment, and this has been very helpful before deploying to Maven. Several times the tests have worked locally but not in a different environment. GitHub sends feedback when the tests fail. When deploying the analyser to the Maven repository, tests are run locally before deploying.

[9] https://github.com/
[10] https://docs.github.com/en/actions

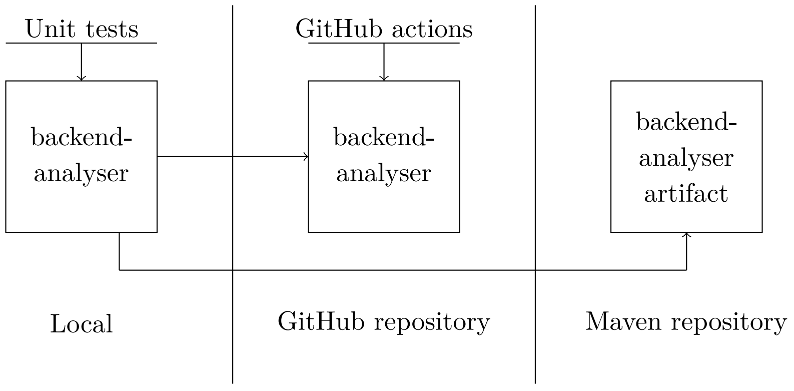
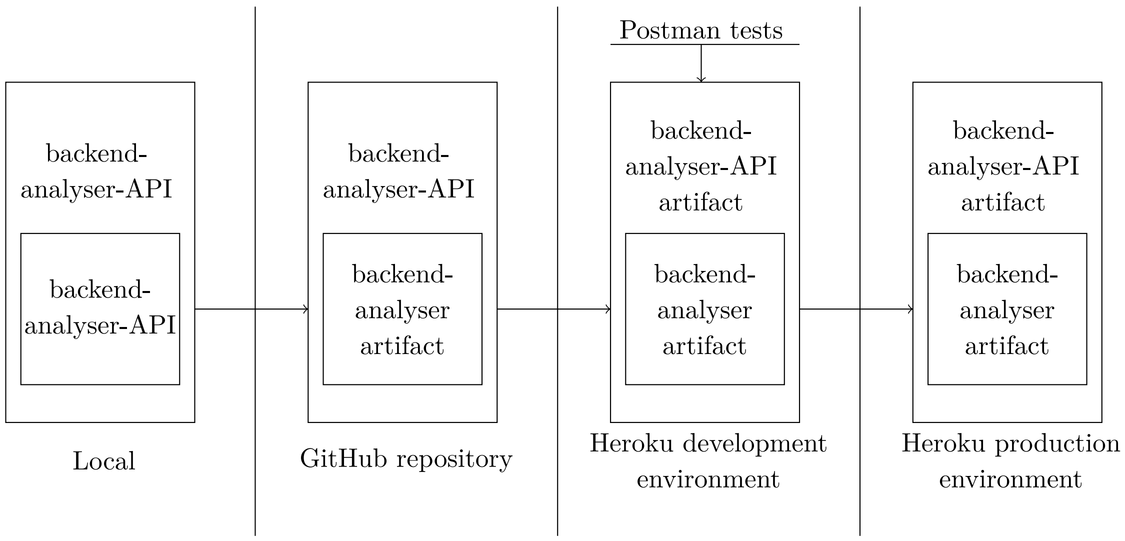


Figure 5.5: Development cycle for master-thesis-backend-analyser

In Figure 5.6, we can see how the API is deployed to Heroku and that it deploys together with the `master-thesis-backend-analyser-artifact` from the Maven repository in Figure 5.5. The different environments are separated by a vertical line, and arrows indicate the flow between the different environments. When we push the code to GitHub, it can be accessed from Heroku and deployed in the development environment. The integration tests are created sending `POST` requests to the API using *Postman*[11]. *Postman scripts* [57] allows us to make assertions based on the response from the API. The integration tests are run before we promote the artifact to the production environment.



Postman tests
backend-analyser-API
backend-analyser-API
backend-analyser-API
backend-analyser artifact
backend-analyser-API artifact
backend-analyser artifact
backend-analyser-API artifact
backend-analyser artifact
Local
GitHub repository
Heroku development environment
Heroku production environment

Figure 5.6: Development cycle for master-thesis-backend-analyser-API

[11]https://www.postman.com

Changelog and semantic versioning

Separating the analyser and API from each other comes with the advantage that we can easily replace the analyser. However, some issues are worth noting when developing these separately. The API does not note changes made to the analyser before the analyser is redeployed and the API has changed the version number for downloading the analyser. Use of *semantic versioning* [58] and keeping a *changelog* makes this issue easier, where we can roll back a non-working version to the latest working version of the analyser. The semantic version follows the pattern `MAJOR.MINOR.PATCH` where a major version is a breaking version, the minor version is a new feature, and the patch version fixes a bug [58]. The need to reflect on the changes and how it affects the usage of the analyser before deploying was a great learning process and helped organise the changelog and version numbers.

5.4 Distribution strategy: Software as a Service or local distribution

To deploy an application as a SaaS has one major benefit: it can be accessed through a browser, making it easily accessible. There are some drawbacks, however: we need internet access to use the application and using remote API's between the client and the server could lead to timing issues due to the transfer of data. Another aspect to consider when creating a SaaS is *scaling*. When many users simultaneously use the application, we need to be able to scale out [50].

Initially we created plugins for both IntelliJ and Eclipse with the desired functionality for the students, with highlighting in the editor to locate the error in addition to display the error message in the source code. IntelliJ and Eclipse are the IDE's the students normally use in INF101, but this semester it was also recommended to use Visual Studio Code. After an informal talk with one of the teaching assistants for the semester, it became apparent that most of the students used Visual Studio Code because they were used to it from the Python course before. Consequently, we discarded the plugins and created a web-application instead. This section discusses some pros and cons with using a plugin to distribute the software locally versus distribute the software as a service, and shows how the core package can be used in a plugin for further work.

For the specifics of the problem where we want to analyse code, we have a problem when using SaaS: the analyser tool is detached from the development environment where the code that should be analysed resides. Unless the students would upload the whole project to the web-page, all the dependencies for the file that is being analysed has to be in the respective file. This would be solved by doing a local distribution, where the users can download a plugin for their IDE, the developer environment, giving access to the project files. In addition to resolving dependencies, the students can interact with the errors in a known environment. Another aspect that is resolved when using a plugin to distribute the software is scaling. However, by deploying the software using PaaS the scaling is handled automatically. A problem with distributing the software locally would be possible security issues if the users do not update when new versions are available, or having to adopt the software to different platforms. Even though the SaaS model have some drawbacks as mentioned for this problem, we created a web-application to evaluate Uncoil such that hopefully more students would want to try it, not having to download anything.

Figure 5.7 shows an ideal architecture of our tool, showing how Uncoil can be extended using both plugins and a web-interface. The core package, `master-thesis-backend-analyser`, could be bundled in an IntelliJ plugin or Eclipse plugin by downloading it with a build tool. A Visual Studio Code plugin would have to use the API, because it is written in JavaScript or TypeScript [59]. An IntelliJ plugin with the architecture as in Figure 5.7 has been created to enable further work and illustrates how the analyser can be used.

JavaParser allows to add a context for resolving dependencies [40], but after an attempt to implement this we discovered some problems causing us to drop the feature. JavaParser takes in a path or jar to where the dependencies resides [40], meaning we would have to create new files containing the dependency code, as well as delete them after analysis. We argue it would be bad practice to have this as a side effect to a plugin, as we would not expect a plugin to save and delete files on the operating system. Possibly a better way to resolve dependencies and allow for several files to be analysed could be to use IntelliJ's PSI (Program Structure Interface)[12] interface to analyse the code, as the same theory in section 4.4 is applicable here as well. However, this is left for further work. The source code for Uncoil is open source (see Appendix A) together with the other code bases for this project. Evaluation of the IntelliJ plugin and creation of the Visual Studio Code plugin is left for further work. We could also have presented the Eclipse plugin, but due to timing constraints this plugin have not been updated to be compatible with the latest version of the analyser.

[12]https://plugins.jetbrains.com/docs/intellij/psi.html

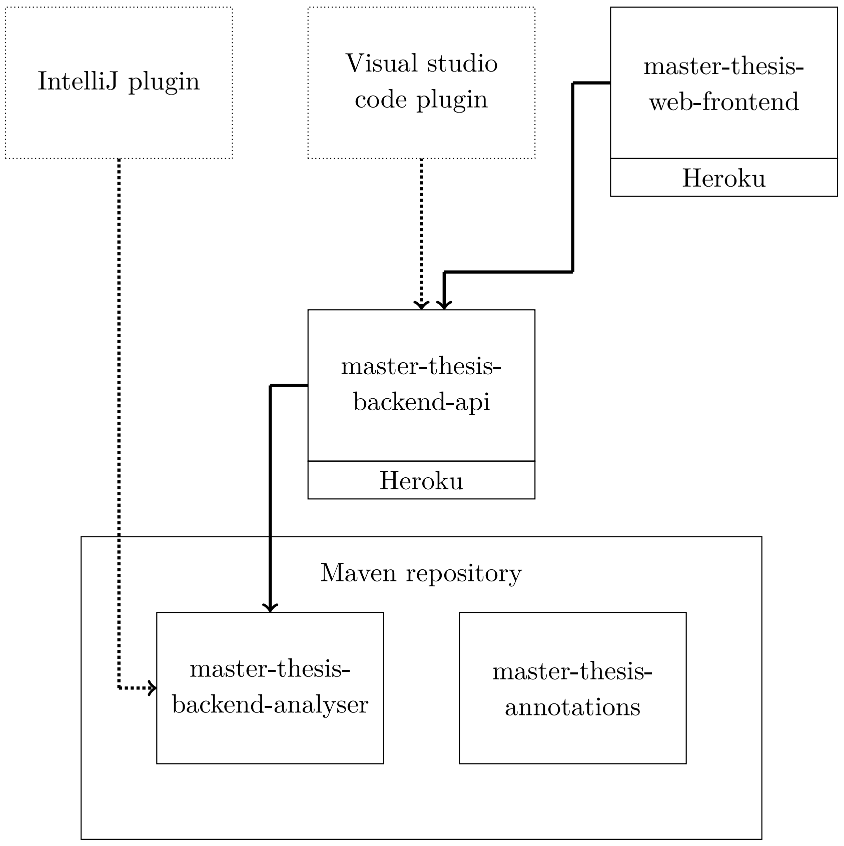

IntelliJ plugin
Visual studio code plugin
master-thesis-web-frontend
Heroku
master-thesis-backend-api
Heroku
Maven repository
master-thesis-backend-analyser
master-thesis-annotations

Figure 5.7: An example of how Uncoil-core can be used

Chapter 6

The evaluation of Uncoil

This chapter aims to answer

RQ1: *What semantic errors do students need help with when transferring from Python to Java?*
RQ3: *How can such a tool help students when transferring from Python to Java?*

---

by letting students try to solve the errors and evaluate Uncoil using a survey. Uncoil was pilot-tested by experienced Java developers to get early feedback. The rest of this chapter represents the evaluation of Uncoil by the students.

6.1 Pilot testing

The pilot testing helped reformulate the questions in the survey and find some bugs with Uncoil. The testers were recruited through informal conversations with co-students and giving them a link with access to Uncoil and project with tasks containing the errors. The testers were given no incentive to participate, but we still had five participants. Overall, the pilot study gave valuable feedback for further developing Uncoil. For example, at the time of the pilot testing we marked $\substack{\text{IFNO}\\\text{BRKT}}$ as an error when there was no brackets, not taking into consideration that it is legal to have one statement in the body without brackets. One of the pilot testers pointed out this issue, further motivating us to focus on the indentation of the body as a way to detect this error.

6.2 A mixed method study to evaluate Uncoil

6.2.1 Method

By letting the students solve the errors and use Uncoil if they have problems, we want to get closer to an answer to RQ1 and RQ3. This mixed method study has an explanatory sequential design [25], where we aim to gain more insight into what makes the errors challenging to tackle and how Uncoil can help with the errors. We gave the students five tasks containing at least one of the errors and asked them to fix the mistake(s) while filling out an anonymous survey. The survey contained some introductory questions followed by questions about the tasks and, in the end, some general questions about Uncoil.

Using a *Likert scale* [60] with a scale from one to five, we measure how much each participant knew Java and Python. By taking the mean values of these answers, we indicate if we have the correct target group to evaluate Uncoil, expecting the mean value for Java to be lower than Python. To indicate if the knowledge level of the students affects the usage of Uncoil, we will define a WEAK student as one who answered less than three on the Likert scale when asked how much they know Java and a STRONG student as one who answered more or equal to three. We hope to get closer to an answer to RQ1 by asking if they had made a mistake before. We use the mode values [61] from these answers as a descriptive statistics method to represent these numbers. We recognise that the group is too small to do a statistical analysis of the data but find the representation of data valuable for a discussion.

The students were asked to solve the errors, use Uncoil if needed, and solve the tasks for themselves. By only using Uncoil when needed, we can indicate how challenging the tasks were, getting closer to an answer to RQ1. We asked the students to fill in the survey after each task to get a more reliable answer, keeping a fresh memory of the mistake. The student is required to put more thoughts into their answers using open ended questions [47], so the survey is created with some long text questions. For each task, the student was asked to explain the error(s) such that we know if the student understood it, which we believe can give us an indication if Uncoil is useful or not, contributing to an answer to RQ3. By inspecting the qualitative answers manually, we aim to get closer to an answer to both RQ1 and RQ3. We categorise the answers to why the students needed help, looking for indications of NST as a reason. To get closer to RQ3, we inspect the answers for why or why not the students found Uncoil helpful.

The students were unsupervised and could perform the study on their computers, adding some limitations to the study. The students were unsupervised primarily due to Covid-19, recruitment issues, and to let the students solve the errors in a known setting without external pressure. We prioritised having an anonymous survey and letting the students explore the errors in a familiar setting, hopefully contributing to getting honest answers from the students. The qualitative answers were designed to act as an explanation for the quantitative answers. Specifically, the follow-up question asked why the student answered the previous question. We believe the quantitative data will be more reliable by getting a reason why the students answered as they did.

Questions

We created questions to collect both quantitative and qualitative data, and the questions are listed in Table 6.1 and Table 6.2, omitting the introductory questions and the question asking what the mistake for each task was. Q1-Q6 are questions asked for each task. Q4 and Q5 were asked for each error for tasks with several errors. The last two questions, Q7 and Q8, are asked to get more general feedback for Uncoil.

Table 6.1 lists the questions that are asked to find out if the error messages are needed. Table 6.2 lists the questions that try to answer if the message was helpful.

	Question	**Answer type**
Q1	Did you use the ”get tips”-button for this task?	Yes or no
Q2	Was the task easy to solve?	Yes or no
Q3	Why or why not was the task easy to solve?	Long answer
Q4	Have you done this mistake before?	Yes, no or maybe

Table 6.1: Questions to to find out if the error message is needed

	Question	Answer type
Q5	Did the message from the ”get tips”-button help you understand the mistake in the code?	Yes or no
Q6	Why or why not did the message help you understand the mistake in the code?	Long answer
Q7	Overall, did you find the messages from the ”get tips”-button helpful?	Yes, no or sometimes
Q8	Why or why not did you find the messages helpful?	Long answer

Table 6.2: Questions to to find out if the error message is useful

6.2.2 Context

The participants were students taking the undergraduate INF101 Java based course at the University of Bergen. The first response to the study was about six weeks into the semester. At that time, according to the lecture plan given by the lecturer that semester they should have learned:

- classes, interfaces, encapsulation, inheritance, composition, abstract classes
- memory locations and how to compare objects
- lists, tables and iterators
- abstraction and modularity, documentation, class diagrams, generics
- test-driven development

6.2.3 Recruitment

From the first response to the last, the study went over eleven days and had seven responses in total. The students were first recruited on 05.02.22 via a letter on the INF101 lecture site, inviting them to join the study. They were told some background information about the study, that the survey was anonymous, how they could join, and how long it was expected to participate. The first recruitment was around three weeks after starting the semester, aligning with Tshukudu and Cutts [6]'s study. The students were asked to send a mail if they wanted to join. While some students were interested, none responded when they got access to the study. They were asked to give their response to the survey within a week.

The first recruitment letter asked the respondents to join via mail because we wanted a small group of students to test Uncoil before recruiting more students for a large scale study. The thought that the students should be recruited by sending a mail was also to allow for a control group if many participated. However, as time went by and none responded to the survey, it became apparent that it would be hard to recruit enough students for a large scale study.

After a week with no responses to the study, the students were motivated to participate before 27.02.22 to win two cinema tickets. The first answer of the study was recorded at 22.02.22. Having only one response by the end of the deadline, the deadline was postponed. They were invited personally during a lecture three weeks after the first recruitment letter to reach more students. They were informed about the project, that their response would be anonymous, that they could win two cinema tickets and that the deadline was by the end of the week, 06.03.22. The lecturers also posted direct links for the study at the lecture site and Discord to participate, so the students did not need to send a mail. The direct posting of links and personal recruitment was the most effective recruitment method, giving six more responses.

It is not clear why so few students answered, but when reminding the first students to answer, it was given an impression that the students had very little time and had a busy schedule. Also, given that many got Covid-19 at that time, it might be that many had low motivation and did not have the energy left to join. Similarly, Brækken [46] also reports on difficulties getting students to join her study.

6.2.4 Setup

There were two setups for the study: one where the participants cloned a repository containing the tasks, ran the code locally, and copied and pasted the code into Uncoil if they needed help. The second setup was in the browser. The code was run from the browser and integrated with Uncoil. The latter was created to lower the threshold for the students to participate after weeks of unsuccessfully trying to recruit students. With Uncoil behaving the same for both setups, we do not believe this makes a significant difference in the results.

The web page the students gained access to when they participated was a very lightweight editor, and Figure 6.1 is a screenshot of Uncoil. By pressing `check code`, the output in the console would tell if they had solved the task or not, accompanied by the output of the executed code. If a hint was needed, the student was told to press `get tip`, and the error message for the task would be printed as shown in Figure 6.1. The student could go back and forth between the different tasks. If they wanted to start over with the tasks, they were asked to reload the page.

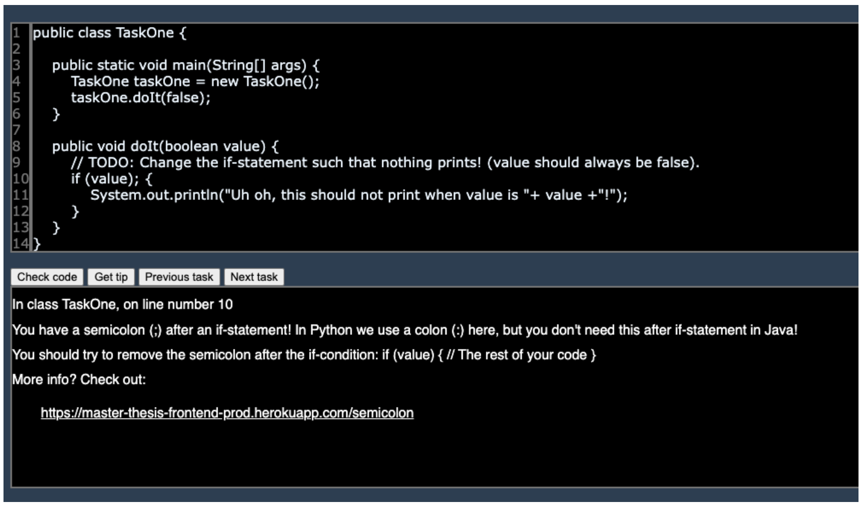
The link that was given to the students was a survey, where they would get access to

Uncoil and answer questions. They also had to consent to participate in the study and that their answers could be used in this thesis. After they had submitted the form, they would get access to register to win two cinema tickets.

For each task, there was at least one error, and the code for the tasks can be found in Appendix C. The errors were distributed as follows:

- **Task one**: $\genfrac{}{}{0pt}{}{\text{IFW}}{\text{SEM}}$
- **Task two**: $\genfrac{}{}{0pt}{}{\text{NOEQ}}{\text{METH}}$ and $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$
- **Task three**: $\genfrac{}{}{0pt}{}{\text{INT}}{\text{DIV}}$
- **Task four**: $\genfrac{}{}{0pt}{}{\text{BIT}}{\text{OP}}$
- **Task five**: $\genfrac{}{}{0pt}{}{\text{IFNO}}{\text{BRKT}}$

The tasks were not randomised.



```
public class TaskOne {

   public static void main(String[] args) {
      TaskOne taskOne = new TaskOne();
      taskOne.doIt(false);
   }

   public void doIt(boolean value) {
      // TODO: Change the if-statement such that nothing prints! (value should always be false).
      if (value); {
         System.out.println("Uh oh, this should not print when value is "+ value +"!");
      }
   }
}
```

In class TaskOne, on line number 10

You have a semicolon (;) after an if-statement! In Python we use a colon (:) here, but you don't need this after if-statement in Java!

You should try to remove the semicolon after the if-condition: if (value) { // The rest of your code }

More info? Check out:

https://master-thesis-frontend-prod.herokuapp.com/semicolon

Figure 6.1: Uncoil with the first task after pressing get tip

6.2.5 Classification of errors

We will make use of the classification of enhanced error messages proposed by Brækken [46], and we will use this classification on the error messages to find out if they are needed or useful. The classifications she proposes are:

- NOT NEEDED
- NEEDED
- NOT USEFUL
- USEFUL

The classifications have been adapted by considering an error message NEEDED if any of the following:

- the task was difficult to solve by more than half of the students
- the mistake has been done before by more than half of the students
- the error message was needed to solve the mistake by more than half of the students

An error message is classified as NOT NEEDED if none of the above applies.

An error message is considered USEFUL if any of the following:

- the error message helped more than half of the students solving the mistake
- the error message helped more than half of the students understand the mistake

An error message is classified as NOT USEFUL if none of the above applies.

An error can be USEFUL, but NOT NEEDED. It can also be NEEDED, but NOT USEFUL [46].

6.2.6 Results

All the answers from the students can be found in Table B.1.

First of all, we want to ensure we have evaluated Uncoil on the target group, namely beginners students in Java who know Python beforehand. To indicate this, we take the mean value of the answers from the questions asking how well they know the different programming languages and expect the mean value for Java to be less than the mean value for Python.

Question	**Mean value**
How well do you know Python? (1=not at all, 5=very well)	3.9
How well do you know Java? (1=not at all, 5=very well)	2.6

Table 6.3: Mean values of students answers for how well they know Java and Python

As expected, we can see that the mean value for Java is less than the mean value for Python, and we have an indication that we have the correct group to evaluate Uncoil.

The questions asking the students what causes the errors allowed us to see if they have understood the error. We found only one unsatisfactory answer to the explanation of the error, and this was for $\frac{\text{BIT}}{\text{OP}}$. The student did not know why it was an error, even though the student used Uncoil to solve the task. However, the student says that Uncoil helped by finding the error and marked it as helpful: "*[...] tool pointed it out very precisely and made me see it*". This student used the local setup of the study.

The answers showed that many of the students had done most of the errors before, and therefore it was easy for them to solve at the time of the study. Consequently, few students needed Uncoil to solve the different tasks. As seen in Figure 6.2, Uncoil was barely used by the students, except for the second task where the $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{NOEQ}\\ \text{METH}}$ errors were present. We note that most of the students here give a reason that they had trouble implementing the equals method.

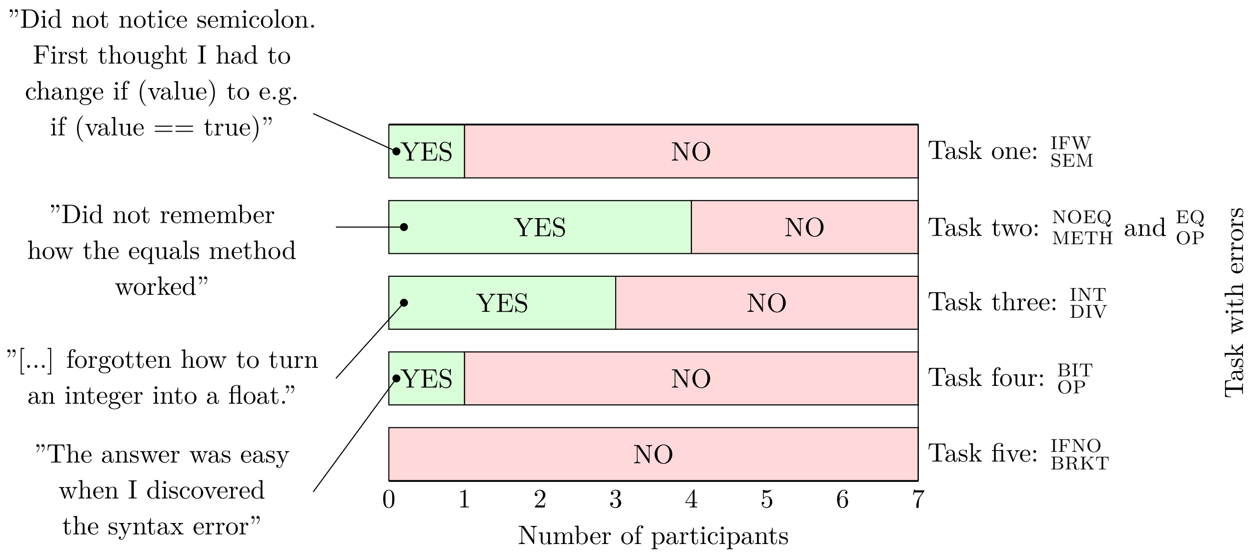


Figure 6.2: Students answers to Q1: *Did you use the "get tips"-button for this task?* and Q3: *Why or why not was the task easy to solve?*. Presentation is inspired by [62]

Even though the students did not need to use Uncoil to solve $\substack{\text{IFNO}\\ \text{BRKT}}$, one student said, "*Provided a better understanding of how Java reads the code.*", showing how Uncoil can be useful even not needed. We also find the same type of answer when looking at the $\substack{\text{BIT}\\ \text{OP}}$ error, where the student who did not need to use Uncoil still found it helpful: "*Checked get tips after I solved the problem, understood more about the di erence between & and &&*".

One student found task one, $\substack{\text{IFW}\\ \text{SEM}}$, difficult, while another student found task four, $\substack{\text{BIT}\\ \text{OP}}$, to be difficult. Three students answered that they thought task two with $\substack{\text{EQ}\\ \text{OP}}$ and $\substack{\text{NOEQ}\\ \text{METH}}$ was hard to solve, and all gave the same reason: they struggled with implementing the equals method. The fourth student in Figure 6.2 that used Uncoil to solve task two said it was not difficult, but had trouble understanding the task. Similar to $\substack{\text{IFW}\\ \text{SEM}}$ and $\substack{\text{BIT}\\ \text{OP}}$, only one student said task three, $\substack{\text{INT}\\ \text{DIV}}$, was difficult to solve. The student tried casting to float first. The other two students in Figure 6.2 that used Uncoil to solve task three reported that it was easy to solve, still one had trouble placing the parenthesis and the other did not remember how to cast to a double.

The splitting of students gave three STRONG students and four WEAK students. Figure 6.3 shows if the different groups used Uncoil to solve the tasks, the STRONG students are shaded. We see that all the students who used Uncoil to solve task one and task two are WEAK students. Two WEAK students used Uncoil to solve task three, while only one STRONG student used Uncoil to solve this task. One STRONG students used Uncoil to solve task four. We have omitted $\genfrac{}{}{0pt}{}{\text{IFNO}}{\text{BRKT}}$ from Figure 6.3 because no student used Uncoil for this task. The STRONG student who needed Uncoil to solve the third task said that float was tried first, while the WEAK students did not remember how to convert to double or place the parentheses.

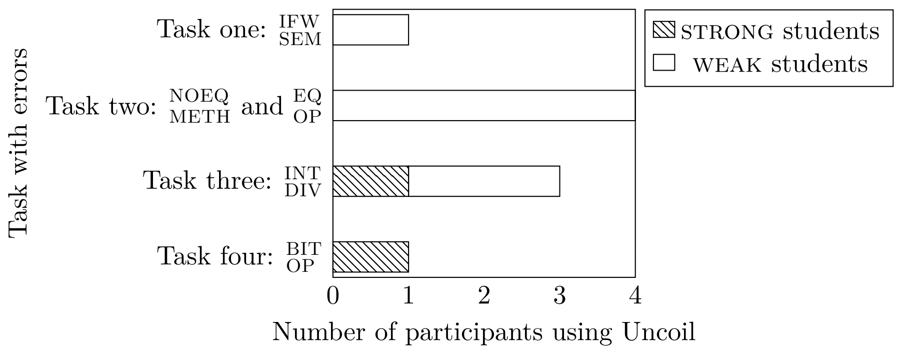


Figure 6.3: Distribution of strong and weak students who used Uncoil

To investigate if any of the students transferred the semantics from Python to Java, causing the error to be difficult, we divide the answers for Q3 into four categories. UNNOTICED: The student did not notice the mistake but solved it after finding it. UNSOLVED: The student saw the error but needed help solving it. TASK: The student had trouble understanding the task. TRANSFER: The student did not think it was a mistake because it reminded them of Python.

Of those students who used Uncoil, we classify the reasons for why in Table 6.4. Note that none of the answers from Q3 can be classified as a possible problem of transfer from Python.

	Unnoticed	Unsolved	Task	Transfer	**Total used the tool**
Task one: $\substack{\textbf{IFW}\\\textbf{SEM}}$	**1**	0	0	0	1
Task two: $\substack{\textbf{NOEQ}\\\textbf{METH}}$ **and** $\substack{\textbf{EQ}\\\textbf{OP}}$	0	**3**	**1**	0	4
Task three: $\substack{\textbf{INT}\\\textbf{DIV}}$	0	**3**	0	0	3
Task four: $\substack{\textbf{BIT}\\\textbf{OP}}$	**1**	0	0	0	1
Task five: $\substack{\textbf{IFNO}\\\textbf{BRKT}}$	0	0	0	0	0

Table 6.4: Categories of why the students who used Uncoil found the tasks difficult.

We will decide if an error is needed mainly by looking at if many participants have done them before and mark an error message as NEEDED if more than half of the students had done it before. Figure 6.4 shows the mode values of Q4, aiming to know if the error has been a previous problem for the students. These numbers are also supported by reasons given by the students explaining why a task was easy to solve, e.g. "*I have struggled with this mistake earlier in the semester [...]*". For every error except $\substack{\text{IFNO}\\\text{BRKT}}$ and $\substack{\text{IFW}\\\text{SEM}}$, we have found a text answer that mentions they have done the mistake before.

For Figure 6.4, on the x-axis, we have the errors together with the mode for that error below. The y-axis is the mode value, which tells how many participants voted for that answer. If the answer is positive, the bar is coloured green. If negative, the bar is coloured red. Generally, if we have a large mode value, the group agrees because many participants voted equally (on the mode). If we have a minor mode or several modes, the group does not agree because they have voted differently. The participants agree that $\substack{\text{EQ}\\\text{OP}}$ has been a mistake they have made before. We also notice that the participants agree that $\substack{\text{IFW}\\\text{SEM}}$ error is a rare error.

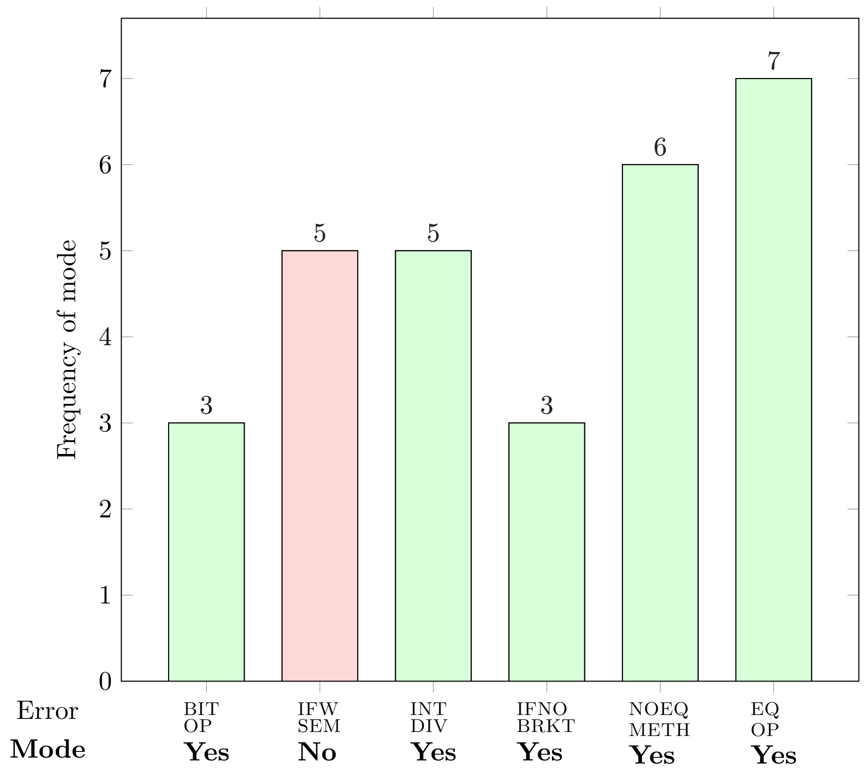


Figure 6.4: Mode values of students answers to Q4: *Have you done this mistake before?*

Generally, if we classify the errors NEEDED or NOT NEEDED based on subsection 6.2.5, we present the table

Error	**Needed**
IFW SEM	No
EQ OP	**Yes**
NOEQ METH	**Yes**
INT DIV	**Yes**
BIT OP	No
IFNO BRKT	No

Table 6.5: Needed or not needed error messages

Figure 6.5 shows how many of the students found Uncoil helpful by splitting the students into two groups, STRONG students are shaded.

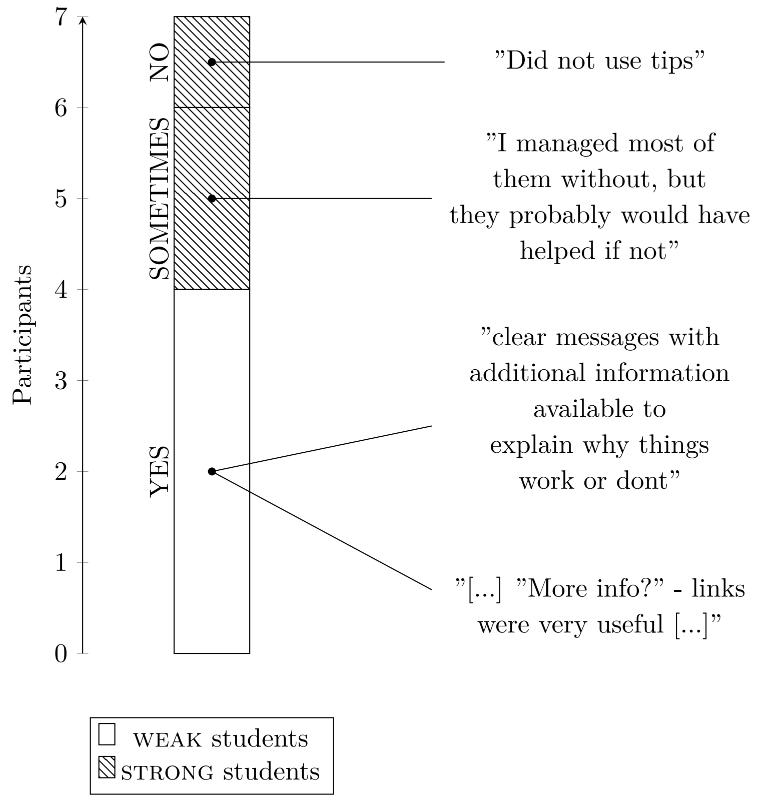


Figure 6.5: Students answers to Q7: ***Overall, did you find the messages from the "get tips"-button helpful?*** **Presentation is inspired by [62]**

The STRONG students did not find the error messages as helpful as the WEAK students. One of the students who said the tips were helpful said: "*The tips explained the problem very well. In particular, "More info?" - links were very useful. It gave a very good understanding of how things worked. Even on the tasks I managed, I could look at the tips and learn even more / get a deeper understanding.*". The main reason why the participants answered that they did not find Uncoil helpful was that they did not use it. We have two responses from where Uncoil was used but reported to be not helpful: $\frac{\text{EQ}}{\text{OP}}$ and $\frac{\text{INT}}{\text{DIV}}$. For the $\frac{\text{EQ}}{\text{OP}}$, the student does not give a reason for why. For the $\frac{\text{INT}}{\text{DIV}}$ the student replied: "*I understood where the error was right away, but needed "get tips" to remember how to change from int to float.*". The students who did not need help with the task containing $\frac{\text{EQ}}{\text{OP}}$ and $\frac{\text{NOEQ}}{\text{METH}}$ are the STRONG students, and the feedback from these students were generally that they had done the mistake before or learnt about it at this time.

Even though it felt natural to explain both $\substack{\text{NOEQ}\\\text{METH}}$ and $\substack{\text{EQ}\\\text{OP}}$ at the same time, this made it harder to analyse the results. Due to inconsistency in the answers to Q5 and the corresponding qualitative question Q6, we discarded the results for Q5. As a consequence, the classification of errors as USEFUL or NOT USEFUL is inconclusive. However, by analysing the results for Q6 we get the impression that Uncoil was helpful even though not needed at the time. One student found Uncoil very helpful for understanding $\substack{\text{EQ}\\\text{OP}}$: "*Made it easier to understand the di erence between == in Java and Python. Better understanding of when to use .equals () and when to use ==.*".

6.2.7 Discussion

Even though most students did not need Uncoil at the time, the main impression is that it is useful for the WEAK students. Furthermore, our results indicate that the students have done most of the mistakes before, so we believe Uncoil would be needed and useful earlier in the semester.

The fact that we saw no indication in Table 6.4 of semantic transfer was unexpected to us, because this contradicts with other similar studies [6], [8]. We believe the results are a cause of having conducted the study later than intended, indicated by students answering that they had made the mistakes earlier in the semester. Tshukudu and Cutts [8] found indications that students transfer the semantics of the two languages two weeks into the semester, and the follow up study [6] found this for three weeks. However, at the similar time our study was conducted, six weeks into the semester, it is found that students had troubles with objects [8]. The STRONG students in our study had no problems with $\substack{\text{NOEQ}\\\text{METH}}$ and $\substack{\text{EQ}\\\text{OP}}$, while the WEAK did, supporting Tshukudu and Cutts [8] that students have problems with objects.

$\substack{\text{NOEQ}\\\text{METH}}$ is a new concept for the students, so the result for this error is somewhat expected. The students who did not need help with this task were the STRONG students, indicating that the error messages might be useful for students that have trouble following the course. We see this in Figure 6.5 where we split the student into two groups. This is a result similar to Aalvik [63], who also found that the weaker students had more use of the VisAST tool than the stronger students. The results for $\substack{\text{EQ}\\\text{OP}}$ being NEEDED are supported across different studies: Rosbach [16] found that the *Wrong condition* mistakes were the most common among students, Tshukudu and Cutts [6] found students had problem with this error when transferring from Python, Hristova *et. al.* [2] found this error regarding strings, and Brown and Altadmri [15] found that the error regarding strings is the 6th frequent error the students make when ranking Hristova *et. al.* [2]'s list. Comparing the focus group result with the students' responses, we see the same trend when deciding if $\substack{\text{EQ}\\\text{OP}}$ is a frequent error: all participants have either made this mistake or seen it before. We indicate that the students need help with the $\substack{\text{EQ}\\\text{OP}}$ error.

Having found no previous research mentioning the $\substack{\text{NOEQ}\\\text{METH}}$ error, we have less evidence to tell if students need help with this error. Nonetheless, this error was proposed by a lecturer in Java during the focus group due to the experience that Java students had persistent troubles with the equals method throughout the course. The results from our study support this, with six students saying they had made this mistake before in Figure 6.4. The qualitative answers from three students who said they struggled with implementing the equals method also strengthened this result. From this, we indicate that the students need help with the $\substack{\text{NOEQ}\\\text{METH}}$ error. We do not get a clear enough picture of if or how Uncoil helped the students with the $\substack{\text{NOEQ}\\\text{METH}}$ error.

Finding the $\substack{\text{INT}\\\text{DIV}}$ or a variation of it mentioned across three different studies [2], [6], [12] we have some indication that this is a common error, and this is strengthened by our study. The feedback for $\substack{\text{INT}\\\text{DIV}}$ gives an impression that the suggestion helped solve this error, supporting the results from [45]. The less notable errors we think the students need help with are $\substack{\text{BIT}\\\text{OP}}$ and $\substack{\text{IFNO}\\\text{BRKT}}$, with less than half of the students report they have done it before. One student mentioned that the $\substack{\text{BIT}\\\text{OP}}$ error was made before as a text answer, while none of the students mentions this for the $\substack{\text{IFNO}\\\text{BRKT}}$ error.

A surprising finding is that no students used Uncoil to solve the $\substack{\text{IFNO}\\\text{BRKT}}$ error. This was the last task, which might have affected the answers somehow. The text answers also supports this, saying that it was similar to $\substack{\text{IFW}\\\text{SEM}}$ and therefore easy to solve. The randomising of tasks could, in hindsight, be solved by creating a separate page for each task and attaching the respective link to the question in the survey. However, there was no time to conduct the study again and recruit new students, so this is left for further work. We do not believe the lack of randomisation affected Q4 because it does not ask to solve the error. Because the other errors are not similar like $\substack{\text{IFW}\\\text{SEM}}$ and $\substack{\text{IFNO}\\\text{BRKT}}$, we also believe these results were not significantly affected by not randomising the tasks. We know that Rosbach [16] found that $\substack{\text{IFNO}\\\text{BRKT}}$, as a ***Grouping problem***, does not occur that often among the students, aligning with our results from asking if the error was made before. In the focus group, half of the participants had not seen the $\substack{\text{BIT}\\\text{OP}}$ error before, and Brown and Altadmri [15] ranked this as the 12th most frequent error among Hristova ***et. al.*** [2]'s list. This concurs with the findings in this study in that $\substack{\text{BIT}\\\text{OP}}$ is less frequent. Even though both $\substack{\text{BIT}\\\text{OP}}$ and $\substack{\text{IFNO}\\\text{BRKT}}$ are less frequent, being the cause of two real-life errors as mentioned in section 1.1, we have indications that the errors should be focused on when teaching novice Java students.

An interesting finding was that one of the students used the tool to solve $\substack{\text{BIT}\\\text{OP}}$ but could not answer the cause of the error, indicating that the error message did not help in understanding the problem. It is mentioned by [64] that the programmer might read the first part of an error message, look at the code again and try to correct the mistake. This might explain why the student did not know the cause: the student read where the error was and added the extra operator without any further ado. However, this case could indicate that the student found the error message too long or hard to read, agreeing with [64]'s results that natural language errors can be harder to read than errors in code. Nevertheless, the student found the message helpful by answering that it helped find the bug.

Based on the literature review for finding relevant errors we got an indication that $\substack{\text{IFW}\\\text{SEM}}$

would be a relevant error. During our studies, both the result from the focus group and the mixed method study contradicts this assumption and indicate that the $\overset{\text{IFW}}{\text{SEM}}$ error is a rare error. In Brown and Altadmri [15] it was counted the number of occurrences of semicolon after loop structures, giving a larger scope and possibly a higher frequency than $\overset{\text{IFW}}{\text{SEM}}$ alone, so we believe this is the reason for the contradiction. Only one student had trouble with this error because it was UNNOTICED, and five students said they had not made this mistake before in Figure 6.4. Even though we found that the $\overset{\text{IFW}}{\text{SEM}}$ error is not relevant for students, there might be other variations of the error that would be relevant, like a while or for structure, mentioned by the focus group. We indicate that the students do not need help with the $\overset{\text{IFW}}{\text{SEM}}$ error, partly contradicting Hristova *et. al.* [2]'s result.

Since the students barely needed Uncoil, it is difficult to conclude what error messages are helpful. However, we have indications that the students might learn from using it, even though they did not need it at the time.

Chapter 7

Discussion

By doing a literature review, focus group and a mixed method study our collective results suggest that novice Java students need help with the errors $\substack{\text{EQ}\\ \text{OP}}$, $\substack{\text{NOEQ}\\ \text{METH}}$ and $\substack{\text{INT}\\ \text{DIV}}$ at an early stage when learning Java. Some students might also benefit from error messages for $\substack{\text{BIT}\\ \text{OP}}$ and $\substack{\text{IFNO}\\ \text{BRKT}}$ even though we have marked them as not needed in our study. We also indicate that the students do not need help with $\substack{\text{IFW}\\ \text{SEM}}$. We found no indications that the students transfer the semantics from Python to Java at the time of the mixed method study. We found that the errors can be detected by doing semantic analysis on the AST and that bytecode analysis is not suitable for finding these particular errors, but it is likely to be useful in other cases. Uncoil can be distributed as a SaaS to make it easily available or as a plugin presenting the error in a known environment. Based on previous research [45], [46], [47], [48], we developed a template for the error message's structure, containing position, the cause of the error, how it relates to Python, a suggestion to solve the error and a link with more info. Based on our evaluation of Uncoil, we found that it helped some students to gain more knowledge about the difference between Python and Java and that it helped with solving some of the errors using suggestions.

Our results for RQ1 strengthen some of the results for Hristova *et. al.* [2], while it weakens some. First off, we feel confident that the $\substack{\text{EQ}\\ \text{OP}}$ error is a problem for the students and that we should extend the error from only considering string comparisons to include all objects. The latter is also supported by Tshukudu and Cutts [8], finding that students struggle with array equality as well in Java. Some of the errors Hristova *et. al.* [2] suggest students make, like $\substack{\text{IFW}\\ \text{SEM}}$, are weakened by our studies. Considering the results from Brown and Altadmri [15], our results might be off because we have chosen a smaller scope of the error, focusing only on the if-statement. Further research should include other control constructs to get a reliable result. To our surprise our results contradict previous research [6], [8] that suggests students transfers their semantics from Python to Java in a negative way. Our results indicate that none of the students transferred the semantics six weeks into the semester. However, our study was performed three weeks later than Tshukudu and Cutts [6], giving a plausible explanation for the contradicting result. Our results indicate that the students have had previous problems with the errors, in turn supporting Tshukudu and Cutts [6].

To answer RQ2 on how to develop Uncoil, we developed a tool with the desired functionality. We can use Uncoil both as a web application and plugin by providing a core functionality deployed as a Maven dependency. A proper editor is out of the scope of this thesis, but Uncoil can easily be used by other applications like TurtleDuck [65], an online learning environment, or online IDEs such as Replit[1]. We provide a demonstration of Uncoil used by TurtleDuck in Appendix A. Based on previous studies, we focus on the structure of the error message, creating a suggestion and not having a too long error message. We saw in section 4.1 that IntelliJ generates a warning for some of the errors. This provides an alternative for presenting the error messages by highlighting code in the editor to notify the programmer, and shows another downside to developing a web application instead of a plugin. Highlighting was initially implemented in the plugins. Using visual representation of the error, like highlighting, helps improve the response time for the students [47], and highlighting is used in [22] as a way to show the errors.

To answer RQ3, we conducted a mixed method study on seven students, and even though Uncoil was not needed at the time, the primary impression is that it is still useful. Similar to Aalvik [63], we found that the WEAK students found the messages helpful. Another way to evaluate Uncoil and test the errors on students could be to do a pre and post test with the errors and compare the results. This has been done in previous studies [20], [22], to evaluate tools with similar functionality as this tool. Since our study was conducted later in the semester than planned, the pre and post-test study design might not have been ideal because most students did not use Uncoil to solve the tasks. Moreover, recruiting students could be more complex because the study would be more comprehensive. We suggest to do a pre and post-test of Uncoil earlier in the semester for further work.

7.1 Limitations and threats to validity

The focus group was conducted to discover what errors found at the time might be relevant and discover more relevant errors. The problem description was quite general at the time. Afterwards, it was narrowed down to a more specific problem asking how students handle semantic errors when going from Python to Java. Thus, the $\substack{\text{INT}\\ \text{DIV}}$ error was not found until after the focus group, and some errors that were presented to the group were no longer relevant. In an ideal situation, we would run a new focus group when we had more knowledge about the subject and had found more relevant errors. Due to time limitations, there was no time to hold a second focus group. Nevertheless, the results from the group were beneficial to strengthen the results from the mixed method study, where we see that both groups agree that $\substack{\text{IFW}\\ \text{SEM}}$ is not needed, while $\substack{\text{EQ}\\ \text{OP}}$ is needed.

[1] https://replit.com/

Additionally, the focus group helped with eliminating some errors, like the $\genfrac{}{}{0pt}{}{\text{ST}}{\text{OB}}$ and $\genfrac{}{}{0pt}{}{\text{IGN}}{\text{RET}}$ and adding more relevant errors, $\genfrac{}{}{0pt}{}{\text{NOEQ}}{\text{METH}}$ and $\genfrac{}{}{0pt}{}{\text{IFNO}}{\text{BRKT}}$.

After analysing the data from the mixed method study, it is clear that the study should have been done earlier in the semester. Even though students were recruited about three weeks into the semester, none of the students answered until three weeks later. The students were unsupervised during the study because of Covid-19, and the thought that some students would be nervous about meeting in person. Also, we wanted the students to be comfortable and not stressed by being observed when doing the tasks. The students were asked to solve the tasks themselves, but we can not guarantee this since they were unsupervised. One of the students answered ”we” in one of the text answers, but we can not know if this is a way of speaking or a participant that got help. Regardless, if the participant collaborated and still got stuck, we think this speaks in favour that the errors are a problem for the students. However, there should be no motive for the students to not try for themselves because they were informed that the answers were anonymous and had no effect on their grades. The students were asked to fill in the form after trying to solve a task and only use Uncoil for help, but by not supervising the students we do not know if they actually used the tool, or if they gained knowledge elsewhere during the study. By inspecting the qualitative answers though we do get the impression that they indeed used the tool when reporting it.

Because the tasks were given in the browser, some students solved them in an unfamiliar environment. The editor in the browser only contained non-highlighted code and line numbers, causing the mistakes to be more difficult to notice than in a familiar environment. As we saw in Table 6.4 two students had problems with seeing two of the mistakes, and two students reported that it was challenging to place the equals method when implementing it. Interestingly, the participant who did not notice the $\genfrac{}{}{0pt}{}{\text{BIT}}{\text{OP}}$ error used the local setup for the study. Implementing the functionality in a plugin or developing the web prototype further would have eliminated the variable of not having syntax highlighting. It would also allow the whole project to be analysed, possibly making the tasks easier to work with for the students. However, some disadvantages might also arise, like downloading the plugin, and the threshold for participating in the study might be higher. Some students might not feel comfortable with cloning repositories, causing them not to participate. By allowing the students to solve the tasks in the browser we believe we have covered a larger span of the students, especially the WEAKER ones, than we would have done by asking them to set up the tasks locally.

A better way to measure how well a student knows Java would be to collect their grades from the exam, as previous studies on students have done [46], [63]. This was not possible because the deadline for this thesis was before the students got their grades. Additionally, we wanted the study to be anonymous to get honest answers and maintain privacy issues. Because the responses to how much they know each programming language are self-reported, there is some insecurity about the correctness of these answers. The inconsistency of the answers shows that a mixed method was a good choice for this study, unlike other studies that have experienced difficulties using this method to evaluate tools [46]. A qualitative follow-up question to the quantitative question acted as a validator for the quantitative question. For example, where the students have answered if they had made a mistake before, we found this mentioned as a reason why the student thought the task was easy, giving more confidence to the quantitative answers. Vice versa, some of the qualitative answers weakened the quantitative responses, causing us to discard some of the answers.

It is possible that a ”think out loud” study [66] like others [8], [9], [11], [22], [46] have done would give more informative answers, because the qualitative answers in our study are based on the students ability to write. While think out loud studied might have given us more information, they have some limitations in regards to the student’s comfort level and how they express their thoughts and the collection of data can be very time consuming [66]. Even though our unsupervised study design gives more insecurity, we believe it benefited from letting the students solve the errors in a known setting. Some students may feel more comfortable by writing their thoughts instead of saying them, and our belief is that by having an anonymous survey we will get honest answers from the students, not being afraid to give a wrong answer.

In an ideal setting and with more time we would have conducted the study again, making changes as follows:

Separate all errors into different tasks. Having both $\substack{\text{EQ}\\\text{OP}}$ and $\substack{\text{NOEQ}\\\text{METH}}$ in the same task made it challenging to analyse the answers and give a clear result.

Rephrase Q5 to ask whether Uncoil helped solving the error or not. Rephrasing this question would have made the results clearer.

Randomise the errors. It is hard to tell if the results for $\substack{\text{IFNO}\\\text{BRKT}}$ is caused by being the last task.

Control group. If we had enough students it would be ideal to set up a control group to compare the results.

Conduct the study in a familiar environment to the students. The tasks were given in a browser with no highlighting. Therefore we may have false positives if Uncoil was needed.

Conduct the study earlier. Based on the answers the students did not have troubles with the errors this far in the semester, so an earlier study can give a more accurate picture of whether the errors are needed.

Unfortunately, there was not enough time to conduct a new study, and the results of this study already show that the students know the errors at this time. Therefore, a new study on Java novices, preferably with the corrections mentioned above implemented, is left for further work. Even though the mixed method study has several flaws, it provides valuable insight in how the students think when they solve the errors and if the error messages provided helped them. Furthermore, we chose to present the study with the corrections as guidelines for further work. To add up for the flaws in the mixed method study, we see that the focus group and mixed method group participants generally agreed about the different errors, strengthening our result. Additionally, the qualitative answers from the students build more confidence in the quantitative data.

Chapter 8

Related work

Looking at previous work [6], [8], [9], [10], [11] on how students or programmers transfer to a new programming language has been crucial to assuming that students experience problems with the transfer. To investigate this topic, Tshukudu and Cutts [8] and Fix and Wiedenbeck [11] have performed experiments where they ask students to choose different options to pair code or solve a task, with some of the options containing concepts from the previous language. Shrestha *et. al.* [17] have examined StackOverflow posts to get closer to how programmers transition to a new language. Tshukudu and Cutts [6], as a follow-up study to [8], asked students to output the result from code examples in Python and Java. Both [8] and [6] found that the students transfer semantics from Python to Java at an early stage and that this happens for array equality, string coercion, string multiplication and integer division. Tshukudu and Cutts [8], Scholtz [9], Fix and Wiedenbeck [11] records the students screen or ask them to think out loud, or both, when the students solve a problem. Unlike the studies mentioned examining NST between programming languages, we ask the students to solve tasks with existing bugs. In the first experiment by Brækken [46], though not aiming to find semantic transfer, she also let students solve errors to evaluate her tool. Like the latter study's first experiment, we let the students solve the errors unsupervised and instead of thinking aloud, we collect text answers. This was because of the difficulty of recruiting students and Covid-19, but we also believe our study benefits from not putting the students in an unknown setting.

When looking at how [9], [10], [11], [20], [21], [22], [46], and [67] have evaluated a tool on students, we see that mainly qualitative methods like screen recording and thinking out loud protocol studies have been used. Holvitie *et. al.* [20] and Shrestha *et. al.* [22] have performed a pre and post test to evaluate ViLLE and Transfer Tutor, respectively. Brækken [46] did both a mixed method study and a think out loud study, while Rigby and Thompson [67] did a mixed method study to evaluate Gild. Unlike in our mixed method study, Brækken [46] found that the qualitative data in the mixed method study was hard to obtain. This might be due to a different study design. While Brækken

[46] asked for feedback from the students in the tool, we asked for this in a separate survey. Nevertheless, a major benefit of giving feedback integrated with the tool is the possibility of recording if the students had used the tool or not, eliminating some of the uncertainty in our study. Additionally, Brækken [46] had more students in the mixed method study and therefore allowed for a control group. Similar to our mixed method study, Krpan *et. al.* [21] used a short anonymous online form to measure what the students thought of their tool. As mentioned in chapter 7, the study design of a "think out loud" was not chosen due to Covid-19, getting honest answers from the students and the time-consuming task of collecting data.

To create Uncoil, we used a static analysis tool, and this has also been done to find similar errors by Hristova *et. al.* [2] creating Expresso and Flowers and Carver [19] creating Gauntlet. We tried without success to download and test Expresso and Gauntlet. The papers [2], [19] were written in 2003 and 2004, respectively so this is perhaps not unexpected. Uncoil has been created with ease of access in mind by being available in the browser and as a plugin. Uncoil can easily be further developed using the tool's core, a Maven package. Unlike Uncoil, Expresso and Gauntlet, one paper has used dynamic template matching models [18] to give feedback to students on semantic errors. Another [11] has used artificial intelligence to give solutions to what the code should do explained in a natural language for the students. Given that both Expresso and Gauntlet used static analysis to find some of the errors in this thesis, we adopted a similar approach.

While Brækken [46] enhance existing error messages and Hristova *et. al.* [2] and Flowers and Carver [19] provide error messages for both syntax and semantic errors, Uncoil focuses on the transfer from Python to Java. We have some tools [11], [20], [22], [21], that focus on transfer between programming languages, while only one of them, ViLLE, from Holvitie *et. al.* [20] can transfer from Python to Java. ViLLE allows the student to see a program in different languages side by side and observe the behaviour. It does not appear to inspect the code when the student writes the code but provides a tutorial. Uncoil is used during development to help the students, even after a tutorial. We believe that by pointing out the error while the student experiences it, they will learn it better. Additionally, even though a student can observe the different behaviours using ViLLE, it might not be apparent to the student why it behaves differently.

Shrestha *et. al.* [22] created Transfer Tutor to help transfer from Python to R. Similar to Uncoil, Transfer Tutor only focuses on one error at a time. However, Transfer Tutor also points out the positive transfer from Python to R. We did not prioritise the positive transfer in this thesis due to previous research [6], [8] indicating this is not a problem for the students. Both ViLLE and Transfer Tutor allow the student to view the new and old language code side by side for comparison. While this was a feature considered when developing Uncoil, we prioritised creating a suggestion on how to solve the error. Different from the others mentioned, Krpan, Mladenovic and Zaharija [21] developed Snap! to transfer from a block-based language to C or Python by translating the block to the target language. In the early phases of this thesis, a similar approach was considered.

However, it was discarded due to similar technology already existing[1] and the thought that students would learn less Java if they could transfer the Python code. Fix and Wiedenbeck [11] created ADAPT (Ada Packages Tool) to help transfer from a procedural language to Ada.

[1]https://www.jython.org/

Chapter 9

Further work

Based on the findings in this study, we propose the following further work:

- Conduct the mixed method study with the corrections as discussed in chapter 7. The main factor would be to do the study earlier in the semester. It could also be done a pre and post test to evaluate Uncoil.
- Evaluate the plugin in IntelliJ created in this thesis.
- Use Uncoil-core to create other plugins.
- Develop Uncoil-core to detect more errors and ignore all the false positives. For example, interesting semantic errors could be the dangling else problem as an extension of the $^{\text{IFNO}}_{\text{BRKT}}$, or the overflow and wraparound of numbers in Java.
- Continue researching the error messages found relevant in this study. More parameters can be looked at, for example, time to fix and frequency of the errors.
- Create a similar tool for experienced Java developers, possibly integrated with GitHub. One possibility could be to analyse the code of a pull request before allowing merge or before promoting it to production. Alternatively, adapt Uncoil-core to use different strategies for the error messages based on different user groups.

As a contribution to further research we provide two datasets from the focus group and an evaluation of Uncoil, a Maven package and API to analyse source code, a collection of semantic errors relevant to students and at last the source code Appendix A. For further work, it is recommended to download the Maven package: `master-thesis-backend-analyser` that contains the core functionality of the application. Documentation is found in the repositories at GitHub at Appendix A.

Chapter 10

Conclusion

The initial questions asked were:

RQ1: *What semantic errors do students need help with when transferring from Python to Java?*

RQ2: *How can we develop a tool to automate feedback for semantic errors when transferring from Python to Java?*

RQ3: *How can such a tool help students when transferring from Python to Java?*

By performing a literature review in section 3.2 and a focus group in section 3.3, we found relevant errors to detect when developing Uncoil. In section 4.1 we found some limitations to existing tools, lacking warnings for errors we believe is important for a novice Java student. To fill the gaps found in section 4.1 we have investigated how to detect the errors both at the abstract level in section 4.4 and at the concrete level in chapter 5. By gaining knowledge from previous research in section 4.5 we got guidelines for how to present the error message to the students. Doing a mixed method study in chapter 6 we aimed to evaluate if the students needed help with the errors found and if Uncoil could help them.

RQ1. We indicate that the errors $\genfrac{}{}{0pt}{}{\text{EQ}}{\text{OP}}$, $\genfrac{}{}{0pt}{}{\text{NOEQ}}{\text{METH}}$ and $\genfrac{}{}{0pt}{}{\text{INT}}{\text{DIV}}$ would be worth focusing on in further research and should be addressed when learning Java. Even though not classified as NEEDED, we would recommend to detect $\genfrac{}{}{0pt}{}{\text{IFNO}}{\text{BRKT}}$ and $\genfrac{}{}{0pt}{}{\text{BIT}}{\text{OP}}$ as well. We found little indication that the students needed help with the $\genfrac{}{}{0pt}{}{\text{IFW}}{\text{SEM}}$, but the students might need help with other variations of this error. We found no evidence that the students transfer the semantics from Python to Java at the time of the study.

RQ2. We have seen that the feedback for semantic errors from Python to Java can be automated by using a static analyser and provided a tool to do this. When presenting the error message, we focus on structure, creating a suggestion and providing more info in a link. Uncoil is developed with a core functionality as a Maven package and is therefore easily distributed and accessible for usage and further development. Additionally Uncoil is accessible through a web service, making it easily integrated into online tools.

RQ3. Our impression is that the students found Uncoil helpful, even though most of them did not need it when the study was conducted. We note that the use of suggestions in the error message and more info available seems helpful for the students.

Our study suggests that the students do not struggle with Negative semantics transfer from Python to Java later in the semester, but they still need help with some of the errors. Based on our findings, we believe novice Java students who know Python will find Uncoil helpful at an early stage when learning Java to get help with the errors.

Bibliography

[1] A. H. Bagge and V. Zaytsev, “Languages, models and megamodels,” in *Post-proceedings of SATToSE 2014*, V. Zaytsev, Ed., vol. 1346. CEUR, 2015, pp. 132–143.

[2] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, “Identifying and correcting Java programming errors for introductory computer science students,” *ACM Sigcse Bulletin*, vol. 35, pp. 153–156, 01 2003.

[3] P. Ducklin, “Anatomy of a “goto fail” - Apple’s SSL bug explained, plus an unofficial patch for OS X!” https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/, 2014, Accessed: 2022-01-12.

[4] A. T. Samsonsen, “Kunne ikke logge inn i Chrome OS på grunn av kodefeil på ett enkelt tegn,” *digi.no*, 2021, Accessed: 2022-03-31. [Online]. Available: https://www.digi.no/artikler/kunne-ikke-logge-inn-i-chrome-os-pa-grunn-av-kodefeil-pa-ett-enkelt-tegn/512109

[5] N. Jiang, “Semantic Transfer and Its Implications for Vocabulary Teaching in a Second Language,” *The Modern Language Journal*, vol. 88, pp. 416 – 432, 08 2004.

[6] E. Tshukudu and Q. Cutts, “Understanding Conceptual Transfer for Students Learning New Programming Languages,” 08 2020, pp. 227–237.

[7] D. J. Armstrong and B. C. Hardgrave, “Understanding Mindshift Learning: The Transition to Object-Oriented Development,” *MIS Quarterly*, vol. 31, no. 3, pp. 453–474, 2007.

[8] E. Tshukudu and Q. Cutts, “Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices,” 06 2020, pp. 307–313.

[9] J. C. Scholtz, “A study of transfer of skill between programming languages,” Ph.D. dissertation, The University of Nebraska – Lincoln, 1989.

[10] K. Walker and S. Schach, “Obstacles to Learning a Second Programming Language: An Empirical Study,” *Computer science Education*, vol. 7, pp. 1–20, 01 1996.

[11] V. Fix and S. Wiedenbeck, “An intelligent tool to aid students in learning second and subsequent programming languages,” *Computers Education*, vol. 27, pp. 71–83, 09 1996.

[12] I. T. Chan Mow, “Analyses of Student Programming Errors In Java Programming Courses,” *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3, pp. 740 – 749, 2012.

[13] P. Jegede, E. Olajubu, A. Ejidokun, and I. Elesemoyo, “Concept–based Analysis of Java Programming Errors among Low, Average and High Achieving Novice Programmers,” *Journal of Information Technology Education: Innovations in Practice*, vol. 18, pp. 49–59, 06 2019.

[14] J. Jackson, M. Cobb, and C. Carver, “Identifying Top Java Errors for Novice Programmers,” in *Proceedings Frontiers in Education 35th Annual Conference*, 11 2005, pp. T4C – T4C.

[15] N. C. C. Brown and A. Altadmri, “Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs,” *ACM Transactions on Computing Education*, vol. 17, no. 2, pp. 1–21, Jun. 2017.

[16] A. H. Rosbach, “Novice difficulties with language constructs,” Master’s thesis, University of Bergen, 2013. [Online]. Available: https://hdl.handle.net/1956/7167

[17] N. Shrestha, C. Botta, T. Barik, and C. Parnin, “Here we go again: why is it difficult for developers to learn another programming language?” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 691–701. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380352

[18] M. Razali, S. Suhailan, M. Mohamed, and M. Sufian, “Online Programming Semantic Error Feedback using Dynamic Template Matching,” *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 9, 2021, Accessed: 2021-12-01. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2021.0120936

[19] T. Flowers, C. Carver, and J. Jackson, “Empowering students and building confidence in novice programmers through Gauntlet,” in *34th Annual Frontiers in Education, 2004 (FIE 2004)*, 11 2004, pp. T3H/10 – T3H/13 Vol. 1.

[20] J. Holvitie, T. Rajala, R. Haavisto, E. Kaila, M.-J. Laakso, and T. Salakoski, “Breaking the Programming Language Barrier: Using Program Visualizations to Transfer Programming Knowledge in One Programming Language to Another,” in *2012 IEEE 12th International Conference on Advanced Learning Technologies*, 2012, pp. 116–120.

[21] D. Krpan, S. Mladenovic, and G. Zaharija, “Mediated Transfer from Visual to High-level Programming Language,” 05 2017.

[22] N. Shrestha, T. Barik, and C. Parnin, “It’s Like Python But: Towards Supporting Transfer of Programming Language Knowledge,” in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018, pp. 177–185.

[23] M. L. Scott, *Programming Language Pragmatics*, 4th ed. Amsterdam: Morgan Kaufmann, 2016.

[24] T. O.Nyumba, K. Wilson, C. J. Derrick, and N. Mukherjee, “The use of focus group discussion methodology: Insights from two decades of application in conservation,” *Methods in Ecology and Evolution*, vol. 9, no. 1, pp. 20–32, 2018, Accessed: 2022-05-30. [Online]. Available: https://besjournals.onlinelibrary.wiley.com/doi/abs/10.1111/2041-210X.12860

[25] J. Schoonenboom and R. B. Johnson, “How to Construct a Mixed Methods Research Design,” *Kolner Zeitschrift Fur Soziologie Und Sozialpsychologie*, vol. 69, no. Suppl 2, pp. 107–131, 2017, Accessed: 2022-05-30. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5602001/

[26] M. S. Khoirom, M. Sonia, B. Laikhuram, J. Laishram, and T. D. Singh, “Comparative Analysis of Python and Java for Beginners,” vol. 07, no. 08, p. 24, 2020.

[27] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[28] D. Crockford, “The application/json Media Type for JavaScript Object Notation (JSON),” Internet Requests for Comments, RFC Editor, RFC 4627, July 2006. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4627.txt

[29] *Python 3.10.4 documentation*, Accessed: 2022-05-31. [Online]. Available: https://docs.python.org/3/glossary.html#term-bytecode

[30] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. SMith, *The Java Virtual Machine Specification*, Java SE 18 ed., Accessed: 2022-05-31. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se18/html/index.html

[31] R. Lämmel, *Software languages : syntax, semantics, and metaprogramming*. Cham: Springer International Publishing, 2018.

[32] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification*, Java SE 18 ed., Accessed: 2022-05-31. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se18/html/index.html

[33] M. Zadka and G. van Rossum, “PEP 238 – Changing the Division Operator,” https://www.python.org/dev/peps/pep-0238/, 2001, Accessed: 2022-02-15.

[34] *Python 3.11 documentation*, Accessed: 2022-05-31. [Online]. Available: https://docs.python.org/3.11/index.html

[35] A. Downey, *Think Python*, 2nd ed. Needham, Massachusetts: Green Tea Press,

2015, Accessed: 2022-05-23. [Online]. Available: https://greenteapress.com/thinkpython2/thinkpython2.pdf

[36] J. Kleinberg and Éva Tardos, *Algorithm Design*. London: Pearson, 2014.

[37] P. Calingaert, *Assemblers, compilers and program translation*. London: Pitman, 1979.

[38] J. Zhao, "Static analysis of Java bytecode," *Wuhan University Journal of Natural Sciences*, vol. 6, pp. 383–390, 03 2001.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.

[40] N. Smith, D. van Bruggen, and F. Tomassetti, *JavaParser: Visited – Analyse, transform and generate your Java code base*. Leanpub, 2021, Accessed: 2022-04-20. [Online]. Available: https://leanpub.com/javaparservisited

[41] A. H. Rosbach and A. H. Bagge, "Classifying and measuring student problems and misconceptions," in *Proceedings of Norsk informatikkonferanse (NIK'2013)*, E. Tøssebro and H. Meling, Eds. Trondheim, Norway: Akademika Forlag, 2014, pp. 110–121.

[42] J. S. Ancker, A. Edwards, S. Nosal, D. Hauser, E. Mauer, and R. Kaushal, "Effects of workload, work complexity, and repeated alerts on alert fatigue in a clinical decision support system," *BMC Medical Informatics and Decision Making*, vol. 17, p. 36, Apr. 2017, Accessed: 2022-05-31. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5387195/

[43] L. Murphy, G. Lewandowski, R. Mccauley, B. Simon, L. Thomas, and C. Zander, "Debugging: The good, the bad, and the quirky -a qualitative analysis of novices' strategies," *ACM SIGCSE Bulletin*, vol. 40, pp. 163–167, 02 2008.

[44] M. M. Nicolas Harrand, César Soto-Valero and B. Baudry, "The Strengths and Behavioral Quirks of Java Bytecode Decompilers," 08 2019.

[45] B. A. Becker, "An effective approach to enhancing compiler error messages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 126–131. [Online]. Available: https://doi.org/10.1145/2839509.2844584

[46] S. M. Brækken, "Enhancing Error Messages for Novices in Computer Science Education," Master's thesis, University of Bergen, 2019.

[47] M. Pedroni and B. Meyer, "Compiler error messages: what can help novices?" *ACM SIGCSE Bulletin*, vol. 40, pp. 168–172, 02 2008.

[48] P. Denny, J. Prather, B. A. Becker, C. Mooney, J. Homer, Z. C. Albrecht, and G. B. Powell, "On Designing Programming Error Messages for Novices:

Readability and Its Constituent Factors," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445696

[49] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., February 2015.

[50] I. Sommerville, *Software engineering 10th edition*. London: Pearson, 2016.

[51] K. Fakhroutdinov, "The Unified Modeling Language," https://www.uml-diagrams.org/, Accessed: 2022-05-20.

[52] "HTTP Methods GET vs POST," https://www.w3schools.com/tags/ref_httpmethods.asp, Accessed: 2022-05-02.

[53] K. Brush, "What is a Use Case?" Accessed: 2022-05-02. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/use-case

[54] D. Cohen and P. Costa, "An Introduction to Agile Methods," *Advances in Computers*, vol. 62, pp. 1–66, 12 2004.

[55] A. Martini, T. Besker, and J. Bosch, "Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations," *Science of Computer Programming*, vol. 163, pp. 42–61, 2018, Accessed: 2022-04-19. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642318301035

[56] "DevOps Pipeline | Atlassian," https://www.atlassian.com/devops/devops-tools/devops-pipeline, Accessed: 2022-05-08.

[57] *Scripting in Postman*, Accessed: 2022-05-24. [Online]. Available: https://learning.postman.com/docs/writing-scripts/intro-to-scripts/

[58] "Semantic Versioning 2.0.0 | Semantic Versioning," https://semver.org/, Accessed: 2022-05-08.

[59] "Your First Extension | Visual Studio Code Extension API," https://code.visualstudio.com/api/get-started/your-first-extension, Accessed: 2022-05-08.

[60] W. M. Vagias, "Likert-type scale response anchors," 2006, Clemson International Institute for Tourism Research Development, Department of Parks, Recreation and Tourism Management. Clemson University.

[61] K. J. DeAngelis and S. Ayers, "What Does Average Really Mean? Making Sense of Statistics," *SCHOOL BUSINESS AFFAIRS*, pp. 18–22, 2009.

[62] A. K. Emery, "How to Visualize Qualitative Data," https://depictdatastudio.com/how-to-visualize-qualitative-data/, September 25. 2014, Accessed: 2022-04-17.

[63] R. Aalvik, "VisAST: Generic AST Visualiser for Software Language Education," Master's thesis, University of Bergen, 2019.

[64] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do Developers Read Compiler Error Messages?" in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Buenos Aires: IEEE, May 2017, pp. 575–585. [Online]. Available: http://ieeexplore.ieee.org/document/7985695/

[65] A. H. Bagge. (2022) Turtleduck v0.2.0. Accessed: 2022-05-31. [Online]. Available: https://turtleduck.puffling.no/

[66] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, Jul. 1999, conference Name: IEEE Transactions on Software Engineering.

[67] P. C. Rigby and S. Thompson, "Study of Novice Programmers Using Eclipse and Gild," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, ser. eclipse '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 105–109. [Online]. Available: https://doi.org/10.1145/1117696.1117718

[68] M. Krishna, "What is the cause of NoSuchElementException and how can we fix it in java?" Accessed: 2021-11-02. [Online]. Available: https://www.tutorialspoint.com/what-is-the-cause-of-nosuchelementexception-and-how-can-we-fix-it-in-java

[69] J. Nordfalk, *Objektorienteret programmering i Java*. Globe, 2007.

Appendix A

Source code

Demo plugin For testing the plugin it can either be downloaded from the GitHub links below or use a finished setup with Ubuntu that can be imported using VirtualBox. We recommend to use the VM setup. Go to

`https://universityofbergen-my.sharepoint.com/:u:/g/personal/quk010_uib_no/EeJH_QT9RGJHkKDLRKe3_VsBCQbx2SKShQenYSeyLnqTnw?e=PHdGVx`

and download the .ova file, use the VirtualBox vizard to import it. Start the VM. Password is osboxes.org (same as username). Open IntelliJ, wait till the project has loaded. Go to Tools, press ”Find bugs!”. The error message should appear in the console. Try to comment out different bugs to test the tool. The file needs to be saved after the change before a new analysis.

Demo with TurtleDuck Write some Java code in the editor and press ”Quality”. `https://uncoil.puffling.no/`

Frontend for the tool: `https://master-thesis-frontend-prod.herokuapp.com/`
Tasks for the students:
`https://master-thesis-frontend-prod.herokuapp.com/tasks`

Source code on GitHub

`https://github.com/JennStro/master-thesis-backend-analyser`
`https://github.com/JennStro/master-thesis-backend-analyser-api`
`https://github.com/JennStro/master-thesis-web-frontend`
`https://github.com/JennStro/master-thesis-intellij`

Appendix B

Data from evaluation by students

Question	Student 1	Student 2	Student 3	Student 4	Student 5	Student 6	Student 7
How well do you know Python? (1=not at all, 5=very well)	4	5	4	5	3	3	3
How well do you know Java? (1=not at all, 5=very well)	3	2	3	2	4	2	2
Task 1: IFW SEM How can the text be printed even though "value" is false?	pga syntax-error "if (value);", hvor ; avslutter linjen og koden inni if-statementet blir kjørt uansett om if er true eller ikke, vil jeg anta	printe uten if-statement	Semikolon etter if(value)	Pga semikolon etter if (value); Da forteller du ikke Java hva som skal gjøres når if-statementet blir møtt. Neste linje printer da ut teksten uavhengig av value.	Forid if statementet avsluttes før teksten skrives ut	Fordi semikolonet gjør at System.out.println() ikke er en del av if-setningen.	ensure the if command actually affects the body by removing the semicolon making it separate from the body
Task 1: IFW SEM Did you use the "get tips"-button for this task?	Nei	Nei	Nei	Ja	Nei	Nei	No
Task 1: IFW SEM Did you manage to solve the task?	Ja	Ja	Ja	Ja	Ja	Ja	Yes

Task 1: $\overset{\mathbf{IFW}}{\mathbf{SEM}}$ Was the task easy to solve?	Ja	Ja	Ja	Nei	Ja	Ja	Yes
Task 1: $\overset{\mathbf{IFW}}{\mathbf{SEM}}$ Why or why not was the task easy to solve?	Når jeg oppdaget syntaxfeilen var det enkelt å løse problemet.	har forstått det slik at ";" kun skal brukes i slutten av kodelinjer, men her var ikke if-setningen ferdig	Kjørte først med if(!Value) for å se hva som skjedde, siden dette ikke påvirket print statement måtte feilen ligge i syntaksen et sted. Siden linjen ble printet kunne feilen i syntaksen ikke være inne i krøllparentesene til if-klausulen. Da oppdaget jeg semikolon etter if.	La ikke merke til semikolon. Tenkte først jeg måtte endre if(value) til for eks if (value == true)	Så semikolonet etter if-statementet med en gang	Fordi jeg kjenner til forskjellen i syntax for if-setninger fra Python til Java.	simple one step solution
Task 1: $\overset{\mathbf{IFW}}{\mathbf{SEM}}$ Have you done this mistake before?	Nei	Nei	Nei	Nei	Nei	Kanskje	Yes

Task 1: IFW SEM Did the message from the ”get tips”-button help you understand the mistake in the code?	Nei	Ja	Ja	Ja	Nei	Nei	Yes
Task 1: IFW SEM Why or why not did the message help you understand the mistake in the code?	Jeg brukte ikke verktøyet	sjekka den etterpå og den forklarte godt	Jeg nyttet jo ikke get tips før jeg løste oppgaven, men tipset er svaret på oppgaven.	Fikk bedre forståelse av hvordan Java leser semikolon	Fordi jeg ikke trengte den	Jeg brukte den ikke.	it was clearly formulated, but i did not need it for this instance

Task 2: NOEQ METH/EQ OP How can the condition ”bag1 == bag2” be false? What method do you need to implement in the class Bag?	bag1 og bag2 er objekter og == spør om de er det samme objektet, som ikke stemmer. En egen equals() metode må implementeres i Bag-klassen for å sammenligne feltvariablene i Bag-objektene og bedømme ut ifra det om de er like.	== sjekker om bag1 og bag2 er lagret på samme plass i stack/heap, derfor må vi bruke equals-metoden	Equals-metode må iplementeres. De er to objekter som holder samme verdi, men de har ulik referanse.	== sjekker om minneadressene er like, altså om bag1 og bag2 er det samme objektet og ikke om de har samme innhold. For å sammenligne innhold trenger vi .equals() metoden.	Fordi skjekker om de er det samme ikke om de er lik	equals()	== checks memory location instead of content. equals. needs to be implemented
Task 2: NOEQ METH/EQ OP Did you use the ”get tips”-button for this task?	Nei	Ja	Nei	Ja	Nei	Ja	Yes
Task 2: NOEQ METH/EQ OP Did you manage to solve the task?	Ja	Ja	Ja	Ja	Ja	Ja	Yes

Task 2: **NOEQ METH/EQ OP** Was the task easy to solve?	Ja	Nei	Ja	Nei	Ja	Ja	No
Task 2: **NOEQ METH/EQ OP** Why or why not was the task easy to solve?	Det var litt krøkkete å finne ut hvor jeg skulle skrive, men ellers har jeg lært masse om akkurat dette i INF101 så langt i semesteret, så det var lett å komme på at dette var løsningen.	Husket ikke hvordan equals metoden funket	En del øving med equals-metoder.	Forstod at jeg måtte bruke .equals(), men var ikke sikker på hvordan jeg skulle implementere metoden.	Slet med denne feilen tidligere i semesteret så så med eng gang hva som var feil	Den var grei å løse så fort jeg forstod hva som ble etterspurt.	i failed to put the equals method inside the bag class, took a while to figure that out
Task 2: **NOEQ METH/EQ OP** Have you done this mistake before? (Not implementing equals method)	Ja	Ja	Ja	Ja	Kanskje	Ja	Yes

Task 2: $^{\textbf{NOEQ}}_{\textbf{METH}}/^{\textbf{EQ}}_{\textbf{OP}}$ Have you done this mistake before? (Using == to compare objects)	Ja	Ja	Ja	Ja	Ja	Ja	Yes
Task 2: $^{\textbf{NOEQ}}_{\textbf{METH}}/^{\textbf{EQ}}_{\textbf{OP}}$ Did the message from the ”get tips”-button help you understand the mistake in the code? (Not implementing equals method)	Nei	Ja	Ja	Ja	Nei	Ja	Yes
Task 2: $^{\textbf{NOEQ}}_{\textbf{METH}}/^{\textbf{EQ}}_{\textbf{OP}}$ Did the message from the ”get tips”-button help you understand the mistake in the code? (Using == to compare objects)	Nei	Ja	Ja	Ja	Nei	Nei	Yes

Task 2: $\substack{\text{NOEQ}\\\text{METH}}/\substack{\text{EQ}\\\text{OP}}$ Why or why not did the message help you understand the mistakes in the code?	Brukte ikke verktøyet.	god forklaring		Gjorde det lettere å forstå forskjellen på == i Java og Python. Bedre forståelse for når jeg skal bruke .equals() og når jeg skal bruke ==.	Fordi jeg klarte det uten		it was clear and helped me understand the mistake the problem wanted, still managed to have problems with the brackets on the class
Task 3: $\substack{\text{INT}\\\text{DIV}}$ How can the condition ”a/b == 1.4” be false?	fordi a og b er definert som int og svaret vil derfor også være int. 1.4 er en double, så å endre a og b til doubles løser problemet.	fordi a og b er heltall/int	Integer division runder ned. Løste ved å type.caste a til (double)	a og b er integers, så svaret blir også en integer. For å få desimaltall må vi bruke double.	forid a og be er begge heltall og gir et avrundet heltall som svar	Fordi a og b er begge heltall, og kan dermed ikke ha et svar som er en float.	the typing of a and b are incorrect to match up to the decimal value of 1.4
Task 3: $\substack{\text{INT}\\\text{DIV}}$ Did you use the ”get tips”-button for this task?	Nei	Nei	Nei	Ja	Ja	Ja	No
Task 3: $\substack{\text{INT}\\\text{DIV}}$ Did you manage to solve the task?	Ja	Ja	Ja	Ja	Ja	Ja	Yes

Task 3: $\substack{\textbf{INT}\\\textbf{DIV}}$ Was the task easy to solve?	Ja	Ja	Ja	Ja	Nei	Ja	Yes
Task 3: $\substack{\textbf{INT}\\\textbf{DIV}}$ Why or why not was the task easy to solve?	Enkel løsning på et velkjent problem	visste at int var heltall og da blir også resultatet heltall	Fordi typer og operatorer er gangske innarbeidet.	Forstod at a og b var int og jeg derfor måtte gjøre det om til double først. Litt problem med å sette parantesene på riktig sted, prøvde først double (a) / double (b)	brukte float først som ikke fungerte	Den var enkel å løse, men hadde glemt spesifikt hvordan man gjør om et heltall til en float.	had and solved this problem before
Task 3: $\substack{\textbf{INT}\\\textbf{DIV}}$ Have you done this mistake before?	Ja	Nei	Ja	Ja	Kanskje	Ja	Yes
Task 3: $\substack{\textbf{INT}\\\textbf{DIV}}$ Did the message from the ”get tips”-button help you understand the mistake in the code?	Nei	Nei	Ja	Ja	Ja	Nei	No

Task 3: **INT DIV** Why or why not did the message help you understand the mistake in the code?	Brukte ikke verktøyet	brukte ikke get tips	Jeg brukte ikke tips	Hjalp med å sette parantesene på riktig sted.	Forid jeg brukte float sum ikke virket, men med double som i tipset virket det	Jeg forstod hvor feilen var med en gang, men trengte "get tips" til å huske hvordan man endrer fra int til float.	checked the tip after solving then rebreaking the code and it is clear in telling why it does not initially work
Task 4: **BIT OP** How can we get an ArrayIndexOutOfBoundsException even though we check that the list "strings" is not empty?	Vet ikke helt hvorfor, men vi fikk det ikke når jeg rettet & til &&	and er to "&"-tegn, ikke bare ett	Mangler en "&" for å faktisk kjøre sjekken.	Må bruke && i stedet for &	Fordi && er skrevet med bare en & så skjekker den ikke om begge er true, og siden det er feil i statementet som blir det false	Fordi AND i Java skrives &&. Men litt usikker på hvordan Java tolker koden når man bare bruker &.	& is not the desired operator as it will still check the next component in the if statement regardless of the first component
Task 4: **BIT OP** Did you use the "get tips"-button for this task?	Ja	Nei	Nei	Nei	Nei	Nei	No
Task 4: **BIT OP** Did you manage to solve the task?	Ja	Ja	Ja	Ja	Ja	Ja	Yes

Task 4: $\overset{\textbf{BIT}}{\textbf{OP}}$ Was the task easy to solve?	Ja	Ja	Ja	Ja	Ja	Ja	Yes
Task 4: $\overset{\textbf{BIT}}{\textbf{OP}}$ Why or why not was the task easy to solve?	Svaret var enkelt når jeg oppdaget syntax-feilen	visste at and er det samme som to "&"-tegn	Fordi det var en enkel syntaks-krøll	Har grei kontroll på && og || fra før (men har ikke brukt & i Java)	Så med en gang at det manglet en &	Jeg kjenner til at AND skrives && i Java.	an addition of a single symbol was all that was needed, having done this mistake before it made this easier to spot
Task 4: $\overset{\textbf{BIT}}{\textbf{OP}}$ Have you done this mistake before?	Ja	Nei	Nei	Ja	Kanskje	Kanskje	Yes
Task 4: $\overset{\textbf{BIT}}{\textbf{OP}}$ Did the message from the "get tips"-button help you understand the mistake in the code?	Ja	Ja	Ja	Ja	Nei	Nei	Yes

Task 4: BIT OP Why or why not did the message help you understand the mistake in the code?	Jeg oppdaget ikke at det bare stod & og ikke &&, verktøyet poengterte det veldig presist og fikk meg til å se det.	brukte ikke get tips	Jeg brukte ikke tips	Sjekket get tips etter jeg løste oppgaven, forstod mer om forskjellen på & og &&	Jeg klarte det uten	Jeg brukte den ikke.	it was comprehensive, a bit annoying that i have to break the code each time to check the get tips as it wont show otherwise
Task 5: IFNO BRKT The boolean shouldAddToList is false, so why does it still add numbers to the list?	if statementet manglet *{}*	fordi det er ikke krøllparantes rundt if setningen	Fordi (shouldAddToList) ikke blir evaluert da det mangler krøllparenteser etter forkravet.	Det mangler *{}* for å vise at begge linjene i if-blokken hører til if-statementet.	Siden det ikke er brukt *{}* i if-statementet så blir bare første list.add() del av det, der kjører list.add(2) etter if-statementet	Fordi man ikke har brukt *{}* i den først if-setningen. Da er ikke list.add() en del av if-setningen og kjøres utansett om shouldAddToList er false eller true.	because like in the first task the brackets are wrong, the if statement affects nothing
Task 5: IFNO BRKT Did you use the "get tips"-button for this task?	Nei	Nei	Nei	Nei	Nei	Nei	No
Task 5: IFNO BRKT Did you manage to solve the task?	Ja	Ja	Ja	Ja	Ja	Ja	Yes

Task 5: IFNO BRKT Was the task easy to solve?	Ja	Ja	Ja	Ja	Ja	Ja	Yes
Task 5: IFNO BRKT Why or why not was the task easy to solve?	enkelt å se hva som var feil	prøvde berre å sette inn krøllparantes fordi eg trudde det måtte vara det	Enkel syntakskrøll	Var samme tema som i oppgave 1.	Så det manglet {} med en gang	Fordi jeg visste at man trenger {} rundt en if-setning.	same problem as first task, easy to spot the flaw
Task 5: IFNO BRKT Have you done this mistake before?	Kanskje	Nei	Ja	Ja	Kanskje	Nei	Yes
Task 5: IFNO BRKT Did the message from the "get tips"-button help you understand the mistake in the code?	Nei	Ja	Nei	Ja	Nei	Nei	Yes
Task 5: IFNO BRKT Why or why not did the message help you understand the mistake in the code?	Brukte ikke verktøyet	brukte ikke get tips	Brukte ikke tips	Ga en bedre forståelse av hvordan Java leser koden.	Fant ut av feilen uten	Jeg brukte den ikke.	clear get tips response

Overall, did you find the messages from the ”get tips”-button helpful?	Av og til	Ja	Nei	Ja	Av og til	Ja	Yes
Why or why not did you find the messages helpful?	Jeg har brukt et lignende verktøy i python som var veldig nyttig, dette kan være nyttig hvis man har en kort kode og bare trenger en pekepinn på hva som kan være feil.	godt forklart	Brukte ikke tips	Synes tipsene forklarte problemet veldig godt. Spesielt var ”More info?” - linkene veldig nyttige. Det ga en veldig god forståelse av hvordan ting fungerte. Selv på oppgavene jeg klarte kunne jeg se på tipsene og lære enda mer/få en dypere forståelse.	Jeg klarte de fleste uten, men de hadde nokk hjulpet visst ikke	De var nyttige når man trengte dem. Men flere av oppgavene var såpass lette at man ikke trengte tipsene.	clear messages with additional information available to explain why things work or dont

Do you have any tips or comments on how to improve the tool? (Name suggestions for the tool would be really appreciated!)	Designet kan bli litt finere, sleng på en instruksjonstekst også. Tilbakemeldingen fra verktøyet kunne kanskje vært printet i en egen liten rute så det blir mer oversiktlig.		”Uncoil” er gøy, siden en pyton er en kvelerslange, utviklerne av Python elsker ordspill og tørr humor, og Pythonsyntaks har en tendens til å ha et fast grep på nye java-brukere om de har gått gjennom Python først.			Burde nok gå enda dypere i syntax-forskjeller. Ville trodd at de ulikhetene som presenteres her er ting man bare sliter med helt i begynnelse, men tar veldig fort. Det er andre aspekter ved Java som er vanskeligere å huske på.	in a testing setting allow get tips to still respond with the information after a success message though it should still clearly show that the code worked as intended

Table B.1: Answers from the students

Appendix C

Tasks for the students

```java
public class TaskOne {

    public static void main(String[] args) {
        TaskOne taskOne = new TaskOne();
        taskOne.doIt(false);
    }

    public void doIt(boolean value) {
        // TODO: Change the if-statement such that nothing prints! (value
            should always be false).
        if (value); {
            System.out.println("Uh oh, this should not print when value is
                "+ value +"!");
        }
    }
}
```

Listing C.1: Task one

```java
public class TaskTwo {
    public static void main(String[] args) {
        TaskTwo taskTwo = new TaskTwo();
        taskTwo.doIt();
    }

    public class Bag {

        private int size;

        public Bag(int size) {
            this.size = size;
        }

    }

    public void doIt() {
        // Two equal bags, because they have the same size.
        Bag bag1 = new Bag(5);
        Bag bag2 = new Bag(5);

        // TODO: Change the condition such that the bags are compared
            correctly. Implement the missing method in class Bag above.
        if (bag1 == bag2) {
            System.out.println("bag1 is equal to bag2, good job! :)");
        } else {
            System.out.println("bag1 is not equal to bag2!");
        }
    }

}
```

Listing C.2: Task two

```
public class TaskThree {
    public static void main(String[] args) {
        TaskThree taskThree = new TaskThree();
        taskThree.doIt();
    }

    public void doIt() {
        int a = 7;
        int b = 5;

        // TODO: Change the condition such that "a/b" is equal to 1.4.
        if (a/b == 1.4) {
            System.out.println("Woho, a/b has the correct value!");
        } else {
            System.out.println("Oh no, a/b should be " + 1.4 + " but is " +
                a/b);
        }
    }

}
```

Listing C.3: Task three

```
public class TaskFour {
    public static void main(String[] args) {
        TaskFour taskFour = new TaskFour();
        taskFour.doIt();
    }

    public void doIt() {
        // TODO: Change the if-condition "(strings.length > 0 & strings[0].
            equals("java"))" such that the ArrayIndexOutOfBoundsException
            is avoided. (The list should be empty!)
        String[] strings = new String[]{};
        if (strings.length > 0 & strings[0].equals("java")) {
```

```java
            System.out.println("Uh oh, we should have an empty list!");
        } else {
            System.out.println("Woho! You avoided the
                ArrayIndexOutOfBoundsException, good job! :)" );
        }
    }

}
```

Listing C.4: Task four

```java
import java.util.ArrayList;

public class TaskFive {
    public static void main(String[] args) {
        TaskFive taskFive = new TaskFive();
        taskFive.doIt(false);
    }

    private ArrayList<Integer> list = new ArrayList<>();

    public void doIt(boolean shouldAddToList) {
        list = new ArrayList<>();

        // TODO: Make changes to the if-statement below such that no
            numbers are added to the list. (Both numbers should be added if
             shouldAddToList is true!)
        if (shouldAddToList)
            list.add(1);
            list.add(2);

        // This should □not□ be changed in this task!
        if (list.isEmpty()) {
            System.out.println("Woho, the list is empty!");
        } else {
            System.out.println("This list should be empty, but is " + list)
                ;
        }
    }
}
```

Listing C.5: Task five

Appendix D

Errors presented by Hristova et. al.

Error	**Classification**
= versus ==	Syntax
== versus .equals (faulty string comparisons)	Syntax
mismatching, miscounting and/or misuse of { }, , ( ), “ ”, and ‘ ’	Syntax
&& vs. & and *vs.* &	Syntax
incorrect semi-colon after an if selection structure before the if statement or after the for or while repetition structure before the respective for or while loop	Syntax
wrong separators in for loops (using commas instead of semi-colons)	Syntax
an if followed by a bracket instead of by a parenthesis	Syntax
using keywords as method names or variable names	Syntax
invoking methods with wrong arguments	Syntax
forgetting parentheses after method call	Syntax
incorrect semicolon at the end of a method header	Syntax
leaving a space after a period when calling a specific method	Syntax
>= and =<	Syntax
invoking class method on object	Semantic
improper casting	Logical
invoking a non-void method in a statement that requires a return value	Logical
flow reaches end of non-void method	Logical
methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation	Logical
incompatibility between the declared return type of a method and in its invocation	Logical
class declared abstract because of missing function	Logical

Table D.1: Errors presented by Hristova et. al.

Appendix E

Semantic errors by Chan Mow

Error
non-static method printBools() cannot be referenced from a static context
arrayindex out of bounds exception
java.utils NoSuchElement exception
class' or 'interface' expected
cannot access class; neither class nor source found
system cannot find the path specified

Table E.1: Semantic errors Chan Mow

Appendix F

Reproduction and testing of errors found in literature review and focus group

Error	Reproduction	Error produced
= versus ==	`if (a = 5) {}`	error: incompatible types: int cannot be converted to boolean

== versus .equals (faulty string comparisons)	`Bag bag = new Bag("123");`
	`Bag bag2 = new Bag("123");`
	`if (bag == bag2) { }`
		None

mismatching, miscounting and/or misuse of { }, [ ], ( ), " ", and ' '	`if (bag == bag2) {`	error: reached end of file while parsing

&& vs. & and *vs.* &	`if (args != null & args.`
	`    length == 5)`
		None

incorrect semi-colon after an if selection structure before the if statement or after the for orwhile repetition structure before the respective for or while loop	`if (a != null) ; {return a}`	None
wrong separators in for loops (using commas instead of semi-colons)	`for (int i = 0, i <= 5, i ++)`	error: ';' expected
an if followed by a bracket instead of by a parenthesis	`if {a != null} {}`	error: '(' expected
using keywords as method names or variable names	`int new;`	error: <identifier> expected
invoking methods with wrong arguments	`ArrayList<String> a = new ArrayList<>();` `a.add(1);`	error: incompatible types: int cannot be converted to String
forgetting parentheses after method call	`"123".toString;`	error: not a statement
incorrect semicolon at the end of a method header	`public void test(); { }`	error: missing method body, or declare abstract
>= and =<	`if (5 =< args.length)`	error: > expected

invoking class method on object	`Demo demo = new Demo();`
	`demo.staticMethod();`
		None

invoking non-static method as static	`Demo.nonStaticMethod();`	error: non-static method nonStaticMethod() cannot be referenced from a static context
improper casting	`float f = (5.0/6.0);`	error: incompatible types: possible lossy conversion from double to float
invoking a non-void method in a statement that requires a return value	`"abc".toUpperCase();`	None

flow reaches end of non-void method	`public int method(int a) {`
	`    if (a == 2) {`
	`        return 3;`
	`    }`
	`}`
		error: missing return statement

methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation	`ArrayList<String> a = new`
	`    ArrayList<>();`
	`a.add(String "string");`
		error: ')' expected

incompatibility between the declared return type of a method and in its invocation	`String s = returnsInt();`	error: incompatible types: int cannot be converted to String

class declared abstract because of missing function	`interface IDemo { public void method(); }`
	`class Demo implements IDemo { }`
		error: Demo is not abstract and does not override abstract method method() in IDemo

arrayindex out of bounds exception	`int[] a = new int[5];`
	`a[10] = 5;`
		Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5 at Demo.main(Demo.java:8)

java.utils NoSuchElement exception	`HashSet<String> a = new HashSet<>();`
	`Iterator<String> ai = a.iterator();`
	`ai.next();`
	[68]
		Exception in thread "main" java.util.NoSuchElementException (Shortened for brevity)

class' or 'interface' expected	`int i = 0;`
	`class Demo { }`
		error: class, interface, enum, or record expected

cannot access class; neither class nor source found	`system.out.println("a");` [69]	error: package system does not exist

integer division	`public double method() {`
	`    return 7/5;`
	`}`
		None

if without brackets	`if (condition)`
	`    method1();`
	`    method2();`
		None

no equals method	`class A {}`	None

Table F.1: Reconstruction and testing of errors found in literature review and focus group.

Appendix G

Errors that meet the first criteria

In this thesis, by CE we mean a compile time error. By ERE we mean a runtime error that when the program runs, states exactly what the problem is. For example, that an index is out of bounds error. When we have an Implicit Runtime Error (IRE), we mean that it is a runtime error, but it does not state *exactly* what the problem is.

Error	**CE**	**ERE**	**IRE**	**First criteria met**
= versus ==	Yes	-	-	No
== versus .equals (faulty string comparisons)	No	No	Maybe	**Yes**
mismatching, miscounting and/or misuse of { }, [ ], ( ), “ ”, and ‘ ’	Yes	-	-	No
&& vs. & and *vs.* &	No	No	Maybe	**Yes**
incorrect semi-colon after an if selection struc-ture before the if statement or after the for or-while repetition structure before the respec-tive for or while loop	No	No	Maybe	**Yes**
wrong separators in for loops (using commas instead of semi-colons)	Yes	-	-	No
an if followed by a bracket in-stead of by a parenthesis	Yes	-	-	No
using keywords as method names or variable names	Yes	-	-	No

invoking methods with wrong arguments	Yes	-	-	No
forgetting parentheses after method call	Yes	-	-	No
incorrect semicolon at the end of a method header	Yes	-	-	No
>= and =<	Yes	-	-	No
invoking class method on object	No	No	Maybe	**Yes**
invoking non-static method as static	Yes	-	-	No
improper casting	Yes	-	-	No
invoking a non-void method in a statement that requires a return value	No	No	Maybe	**Yes**
flow reaches end of non-void method	Yes	-	-	No
methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation	Yes	-	-	No
incompatibility between the declared return type of a method and in its invocation	Yes	-	-	No
class declared abstract because of missing function	Yes	-	-	No
arrayindex out of bounds exception	No	Yes	-	No
java.utils NoSuchElement exception	No	Yes	-	No
class' or 'interface' expected	Yes	-	-	No
cannot access class; neither class nor source found	Yes	-	-	No
integer division	No	No	Maybe	**Yes**
if without brackets	No	No	Maybe	**Yes**
no equals method	No	No	Maybe	**Yes**

Table G.1: Error meeting criteria

Appendix H

Focus group recruitment text

Forstår Java studenter semantiske feil?

Universitetet i Bergen Institutt for Informatikk

Er du eller har du vært en gruppeleder for et Javakurs? Har du hjulpet studenter med å finne feil i koden? Eller er du en student som har eller har hatt Java? Hjelp oss å lage et verktøy som kan hjelpe studenter med å finne og forstå semantiske feil i Java!

Vi ønsker å fokusere på semantiske feil, spesifikt feil som både kjører og kompileres uten noen feilmelding.Tidligere studier har vist at disse feilene ikke opptrer like ofte som syntaktiske feil, men når de først oppstår så bruker studentene lang tid på å finne ut av dem.

Hvis DU vil hjelpe oss å hjelpe studenter, del dine erfaring i vår fokusgruppe! Vi trenger 6-8 personer som kan møtes på Universitetet i Bergen for å finne ut hvilke feil vi bør fokusere på, og hvordan best hjelpe studentene med disse feilene. Vi beregner at vi bruker ca. en time på dette.

Hvis du velger å delta, vil du bli spurt om å ta del i en diskusjon hvor målet våres er å bestemme hvilke semantiske feil som er mest relevante for våre studier. Vi har valgt noen feil som vi ønsker å presentere og skal bli diskutert, og du er også velkommen til å foreslå andre feil du tenker kan være relevant.

Det vil bli tatt notater under diskusjonen.

Ta kontakt med Jenny på Jenny.Strommen@student.uib.no for å delta! Din kontaktinfo blir bare brukt til å formidle informasjon om denne fokusgruppen. Kontaktinfo slettes når fokusgruppen er ferdig.

Denne fokusgruppen vil være en del av master-oppgaven: Semantic errors in Java

Veileder: Anya Helene Skrove Bagge anya.bagge@uib.no

Veileder: Anna Maria Eilertsen anna.eilertsen@uib.no

Masterstudent: Jenny Strømmen Jenny.Strommen@student.uib.no

Appendix I

Focus group presentation

Focusgroup

Semantic errors in Java

Plan

- 13:00-13:15

  Introduction
- 13:15-13:45

  Discuss the errors
- 13:45-14:00

  Free discussion, other errors can be added here.

What errors are we looking for?

- Semantic = meaning
- Semantic errors - errors that are syntactically correct, but does not do what we intended to do.
  For example: Use of “==” on objects in Java
- Criteria for including an error:
  Should not give an explicit runtime error or any compile time error.
  Should be detectable by static analysis.

Questions

- Have you seen any cases of this mistake among students?
- Is it likely that an INF101 student makes this mistake?
- Why or why not?
- Do you think a student that knows Python beforehand can easier make this mistake then another student that does not know Python beforehand?
- Why or why not?

Semicolon after if

```
public class Main
{
    public static void main(String[] args) {
        method(0);
        method(5);
    }

    public static void method(int a) {
        int result = 0;
        if (a == 5); {
            result = 5;
        }
        System.out.println(result);
    }
}

// Out
> 5
> 5
```

Semicolon after if

```
def method(a):
    result = 0
    if (a == 5):
        result = 5
    print(result)

method(5)
method(0)

// Out
> 5
> 0
```

Use of bitwise operator instead of logical operators

```python
def checkJava(s):
    return s != None and s.casefold() == ”java”
```

Use of bitwise operator instead of logical operators

```
public boolean checkJava(String s) {
    return s != Null and s.toLowerCase().equals("java");
}
```

And operator Java google search: link

```
public boolean checkJava(String s) {
    return s != Null & s.toLowerCase().equals("java");
}
```

Calling a static method as a normal method

```java
public class Super {
    public static void staticMethod() {
        System.out.println("Super class");
    }
}
```

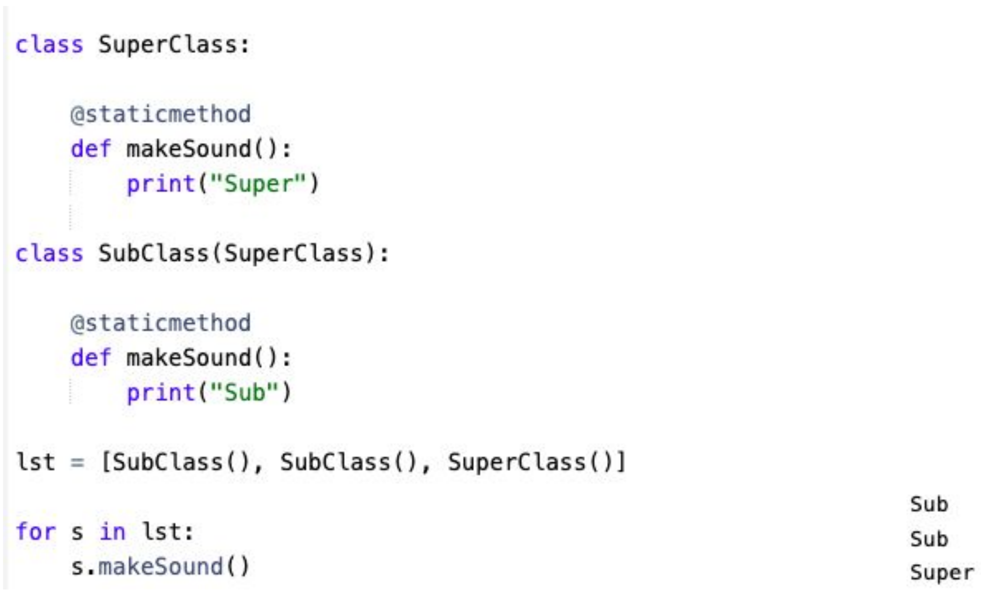
```java
public class Sub extends Super {
    public static void staticMethod() {
        System.out.println("Sub class");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        ArrayList<Super> lst = new ArrayList<>();
        lst.add(new Sub());
        lst.add(new Sub());
        lst.add(new Super());

        for (Super s : lst) {
            s.staticMethod();
        }
    }
}

// Out
Super class
Super class
Super class
```

Calling a static method as a normal method



```python
class SuperClass:

    @staticmethod
    def makeSound():
        print("Super")

class SubClass(SuperClass):

    @staticmethod
    def makeSound():
        print("Sub")

lst = [SubClass(), SubClass(), SuperClass()]

for s in lst:
    s.makeSound()
```

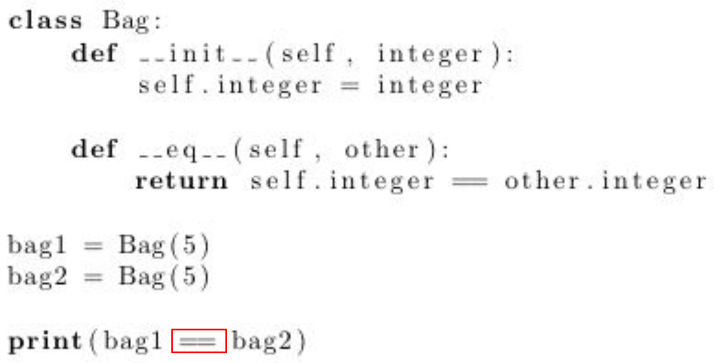
```
Sub
Sub
Super
```

Ignoring return value

```
public int count(String s) {
    s.toLowerCase(); // Return value ignored
    int count = 0;
    for (char ch : s.toCharArray()) {
     if (ch == 'a') {
        count += 1;
     }
    }
    return count;
}

System.out.println(count("AAA")); // 0
System.out.println(count("aaa")); // 3
```

Using == instead of .equals()



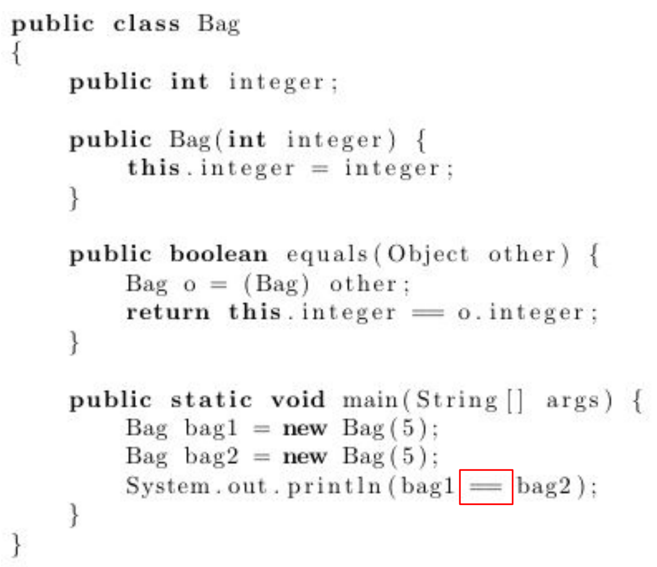
```
class Bag:
    def __init__(self, integer):
        self.integer = integer

    def __eq__(self, other):
        return self.integer == other.integer

bag1 = Bag(5)
bag2 = Bag(5)

print(bag1 == bag2)
```

Using == instead of .equals()



```
public class Bag
{
    public int integer;

    public Bag(int integer) {
        this.integer = integer;
    }

    public boolean equals(Object other) {
        Bag o = (Bag) other;
        return this.integer == o.integer;
    }

    public static void main(String[] args) {
        Bag bag1 = new Bag(5);
        Bag bag2 = new Bag(5);
        System.out.println(bag1 == bag2);
    }
}
```