INSTITUTO
UNIVERSITÁRIO
DE LISBOA

White-Box Assessment for Programming Education

Afonso Manuel Barral Caniço

Master's in Computer Science and Engineering

Supervisor:
Doctor André Leal Santos, Assistant Professor,
Iscte – Instituto Universitário de Lisboa

July, 2024

TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

White-Box Assessment for Programming Education

Afonso Manuel Barral Caniço

Master's in Computer Science and Engineering

Supervisor:
Doctor André Leal Santos, Assistant Professor,
Iscte – Instituto Universitário de Lisboa

July, 2024

witter /ˈwɪt.əʳ/ *v.*

Informal, British. *To talk for a long time about things that are not important.*

Resumo

A testagem de software é maioritariamente realizada em modo de *black-box*, isto é, sem incorporar nos testes qualquer conhecimento relativo ao funcionamento interno de um programa. Esta prática é usualmente suficiente em contexto empresarial ou para desenvolvedores no geral, onde o foco é a produção de resultados fiáveis em que a maioria das tarefas algorítmicas são realizadas por bibliotecas externas. Contudo, para estudantes de programação a um nível introdutório ou similares, pode não ser direto discernir as causas subjacentes de um resultado incorreto num teste ou compreender o incumprimento de certos objetivos no que toca ao funcionamento algorítmico.

Esta dissertação apresenta o Witter, uma biblioteca de testagem de software que permite a educadores de programação definir testes *white-box* para código fonte Java. Os testes analisam a execução e um método contra uma solução de referência, para verificar que não só produz resultados corretos mas também que cumpre o comportamento algorítmico desejado. É detalhada a motivação para o desenvolvimento da biblioteca e descrito esse processo de acordo com os nossos objetivos de investigação.

Avaliamos a eficácia do Witter em avaliar submissões de estudantes para determinar se uma ferramenta de avaliação baseada em eventos na execução do código poderia oferecer informação adicional relativamente a requisitos de comportamento algorítmico incompletos, mesmo quando o código do estudante produz os resultados esperados. Concluímos que uma quantidade considerável de estudantes cometem erros relativos ao comportamento algorítmico das suas implementações, podendo então beneficiar de uma ferramenta de avaliação que fornecesse informação relativa à execução dos seus programas.

Palavras-Chave: *educação de programação*, *testes de white-box*, *avaliação de programação*, *feedback*

Abstract

Software testing is mostly performed in a black-box manner, that is, without incorporating any knowledge of the internal workings of programs into the tests. This practice usually suffices for enterprises and general practitioners, where the focus lies on producing reliable results while most algorithmic tasks are provided by third-party libraries. However, for computer science students and the like, it might not be straightforward to discern the underlying causes of an incorrect test result or to understand why certain algorithmic goals are not met.

This dissertation presents Witter, a software testing library that allows programming instructors to define white-box tests for Java source code. Our tests analyse the execution of a method against a reference solution, to verify that the code not only produces correct results but is also in accordance with a desired algorithm behaviour. We detail the motivation for the development of the library and describe its development process in accordance with our research goals.

We evaluate Witter's efficacy in evaluating student submissions for an introductory university programming course to determine if an assessment tool based on code execution events could offer additional insight regarding incomplete algorithmic behaviour requirements, even when the student's code produces correct outputs. The results support our hypothesis, and we conclude that a considerable amount of students make mistakes related to the algorithmic behaviour of their implemented solutions, and could thus benefit from an assessment tool providing them with information regarding their program's execution.

Keywords: *programming education*, *white-box testing*, *programming assessment*, *feedback*

Contents

List of Figures

List of Tables

List of Acronyms

API: Application Programming Interface

AST: Abstract Syntax Tree

DSL: Domain-Specific Language

ITS: Intelligent Tutoring System

LLM: Large Language Model

CHAPTER 1

Introduction

We begin by contextualising and motivating the research and development work conducted for this dissertation, present the methodology which was employed, and overview the contributions of the work.

1.1. Context and Motivation

Software testing is an integral part of software development, allowing for the assessment of whether or not a given application verifies its intended requirements or produces acceptable results pertaining to the problem domain at hand. By far, the most common method of performing software tests relies on **black-box** testing, where only the outputs of the executed code are considered in the assessment. Extensive work has been conducted on black-box testing libraries and toolkits, of which the most widely known in the realm of Java is the JUnit framework[1]. While such tests suffice for validating the overall functionality of an application and thus are generally sufficient in enterprise environments or for general programming practitioners, they do not provide any means to analyse or evaluate non-trivial aspects of program execution, such as efficiency, memory usage, or algorithm behaviour (that is, if an algorithm executes internally as expected).

It is usually relevant to consider a **white-box** testing method when consideration needs to be given to the internal structure, execution details such as algorithmic behaviour, or otherwise non-trivial program characteristics when testing a piece of software. White-box testing is defined as a software testing methodology that requires knowledge of the internal logic and code structure of a given program, and thus requires the instructor or practitioner executing the testing process to have full knowledge or access to the program's source code [**16**].

Black-box software testing solutions have been extensively studied and some frameworks, such as JUnit, are well-established in the universe of automated software testing. However, while extensive research and development work has also been conducted on white-box automated software analysis and assessment [**26**,**34**,**38**], and most of the existing solutions do effectively allow for the augmentation of the software testing process through a more in-depth code analysis, their thoroughness is usually guaranteed through static code analysis (e.g. Semmle[2]), meta-programming, or similar specialised solutions [**38**]. These techniques, while allowing for an in-depth assessment process, require specialised programming knowledge that may be non-standard or generally misunderstood by regular

[1] `https://junit.org`
[2] `https://semmle.com/`

practitioners; for example, some practitioners might not be familiar with the concept or usage of meta-programming [**15**], which is usually required when augmenting automated assessment systems. This lack of accessibility for general instructors or practitioners often overshadows the usefulness of such solutions in educational applications. As such, this at least warrants research work into the usefulness of an easy-to-use white-box testing solution from the perspective of the instructor or practitioner defining the testing process. An ideal solution would enable in-depth assessment without requiring knowledge of excessively specialised or non-standard programming techniques.

Artificial Intelligence models have become commonplace in the field of programming education, with students relying on ChatGPT or similar Large Language Models (LLMs) to aid their learning process [**37**]. The widespread usage of LLMs by students raises the possibility of augmenting an automated assesment system using LLMs for automated programming assessment. In Lehtinen et al. [**30**], the authors conducted a study on the performance of LLMs in answering questions related to code comprehension, and found that state-of-the-art LLMs such as GPT-4, while able to accurately answer part of the proposed questions, produced errors similar to those of introductory-level students, and had the possibility of "hallucinating" nonsensical explanations in an attempt to support incorrect answers. Moreover, a study by Xue et al. [**47**] finds that the learning performance of introductory programming students is not significantly impacted by the usage of ChatGPT as a learning aid. In fact, students were more likely to ignore other programming aids when they had access to ChatGPT [**47**], which could negatively impact their self-efficacy. Even when it comes to code generation, most LLMs struggle to comply with the best practices and inherent complexity of more advanced topics such as Object-Oriented Programming (OOP), as noted by Cipriano et al. [**11**]. These facts motivate the need for an educational environment which aids students in autonomously improving self-efficacy and program comprehension skills. While our goal is not to discredit the usage of such models as learning aids or assessment tools, we take this as a motivating factor for the development of a deterministic solution to prevent misleading students in their learning process.

When learning a programming language, it is generally regarded that students benefit from informative and constructive feedback that allows them to explore the mistakes in their implementations and thus deepen their understanding [**35**, **46**]. In fact, the absence of constructive feedback can induce the reinforcement of harmful habits; for example, a student might resort to a trial-and-error approach that eventually converges to correct results without contributing to the student's understanding [**2**]. Moreover, it is widely agreed upon that several aspects need to be taken into consideration when grading students' programming assignments. Namely, it is relevant to consider not only the Correctness of a solution, but also its Efficiency and Maintainability [**9**]. While the correctness of an implemented solution is easy to assess through black-box testing, the in-depth assessment of the efficiency and maintainability of a program is not supported by the majority of existing automated assessment systems [**38**]. Another problem is posed by the need to find

the middle ground between feedback that is too vague or unhelpful, and feedback that excessively directs students' learning process. Since students respond more positively when they manage to stay on track and autonomously arrive at the solution to a problem [**28**], it is imperative that the provided feedback offers sufficient guidance without compromising the students' feelings of autonomy. Thus, there is a need for an automated assessment tool designed for educational environments that takes into consideration both the students' learning process and the needs of their instructors.

1.2. Research Questions

The main goal of this dissertation is to explore the practical feasibility of implementing an educational software testing solution that enables a more in-depth analysis regarding the execution of proposed code solutions. We wish to gauge whether or not a library can be easily implemented that allows instructors to define assessment scenarios for student code in an easier, more readily accessible way, and whether students benefit from the more detailed feedback made possible by the library's analysis of the execution of their code, as detailed in Section 1.1. We explore not only the feasibility of implementing such a library, but the effect on its proposed target user base, that is, students and instructors of programming at an introductory university course level. To this effect, this dissertation aims to explore the following research questions:

RQ1 Is it feasible to implement a solution to dynamic white-box software testing?

RQ2 To what extent do introductory programming students' submissions conform to internal behaviour requirements?

1.3. Methodology and Contributions

The content of this dissertation is based on the groundwork laid in the following papers [**6**,**7**]:

Witter: A Library for White-Box Testing of Introductory Programming Algorithms
Afonso B. Caniço, André L. Santos
SPLASH-E, SPLASH'23, October 25-27, 2023, Cascais, Portugal

A Domain-Specific Language for Dynamic White-Box Evaluation of Java Assignments
Afonso B. Caniço, André L. Santos
ICPEC'24, June 27-28, 2024, Lisbon, Portugal

While these papers describe the motivation and the development of our library, described in Section 4.1, this dissertation describes in further detail the library's development, further including features implemented after the paper's publication, as well as a more refined literature review on related work.

This dissertation presents a practical approach at exploring a possible answer to the proposed research questions. We envision an easy-to-use solution to white-box test case definition by instructors, requiring no further knowledge than what is already possessed by

most programmers. As for the automated assessment, our approach relies on the dynamic collection of code execution information to construct what we refer to as *runtime metrics*. The system for capturing runtime information is based on Strudel[3], a library providing an infrastructure for fine-grained observation of code runtime events.

We present Witter[4], a novel software testing library developed with special consideration for educational environments that provides a simple infrastructure for defining and running white-box test cases for Java programs supporting a subset of Java's syntax. Test cases are defined by annotating source code solutions of assignments with simple directives that describe what should be tested. Test case definition using Witter does not require any sort of program instrumentation skills, as the collection of in-depth metrics about the programs' behaviour is expressed with high-level directives. In addition to a flag indicating whether the program passed or failed each defined test case, the output of the tests includes messages concerning mismatches regarding what was expected to happen during execution. Witter was developed from the ground up as the main software artefact of the work presented in this dissertation.

Our aims with the proposed library are twofold. On the one hand, automated assessment systems can be enriched with white-box tests, which will verify with more precision if assignment goals are met. When such tests fail, users will be provided with constructive, formative feedback to improve future submissions. On the other hand, white-box tests could be integrated into assignment development environments (in addition to the usual black-box tests). The test results and feedback messages raise awareness of assignment goals that are not fulfilled, which could otherwise go unnoticed if exclusively black-box testing is applied and the return values are correct. Both of these use cases could assist students in achieving autonomous learning paths toward the desired practice goals.

The development of this dissertation and the aforementioned library follows the Design Science Research [**4**] process outlined in Figure 1.1.

We start by identifying the problem and motivating the need for a novel solution, whose objectives are then defined so that an initial prototype can be developed to show the practical feasibility of the proposed solution. The design and development process, which aims to provide an answer to Research Question **RQ1**, continues and is guided by an evaluation process which uses a dataset of real student submissions for an introductory programming course, which provides an opportunity to answer Research Question **RQ2**.

1.4. Document Structure

We first conduct a literature review and compile a collection of relevant related work in Chapter 2, which includes a motivating review of academic publications on automated assessment in educational environments in Section 2.3, a review of automated white-box-assessment in Section 2.2, and a review on the effect of feedback on introductory-level programming students in Section 2.4, motivating not only the need for a tool which allows

[3] `https://github.com/andre-santos-pt/strudel`
[4] `https://github.com/ambco-iscte/witter`

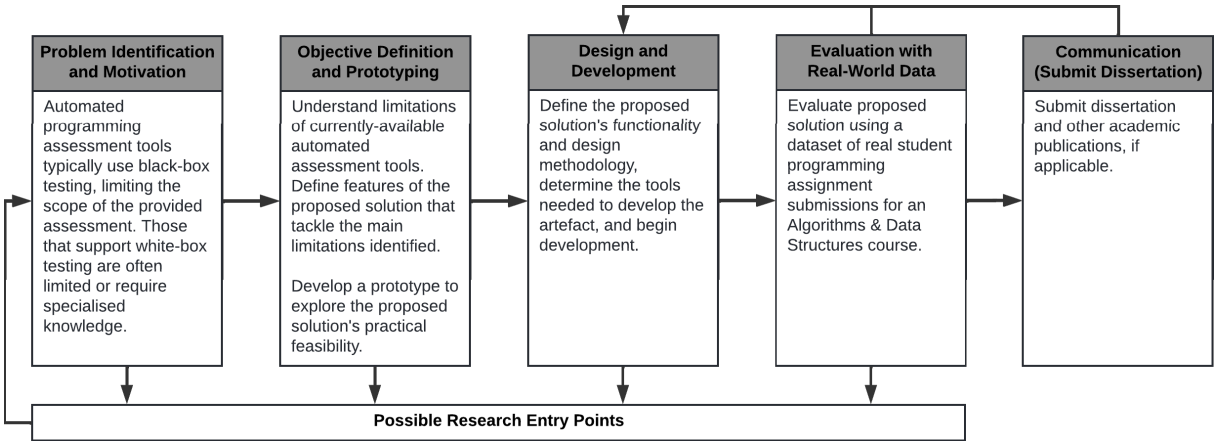

FIGURE 1.1. Design Science Research process diagram.

students and instructors to easily integrate formative feedback in existing assessment and development solutions, but also the potential of a feedback generation tool to positively impact students' learning process. The methodology of our literature review is described in Section 2.1.

Chapter 4 describes our approach regarding the shortcomings identified during the described literature review. Namely, we present our implementation of a white-box assessment library in Section 4.1, describing its usage and giving examples in Section 4.2 and providing a more detailed, technical analysis of the implementation in Section 4.3.

In Chapter 5, we apply our proposed library to a set of introductory programming students' assignment submissions in order to gauge the efficacy of the tool when applied to a context representative of its desired use case as an educational tool. This assessment is contextualised in Section 5.1, with the evaluation method being described in Section 5.2 and the results being presented in Section 5.3.

Finally, in Chapter 6 we discuss the results and limitations of our evaluation and the contributions we have made in regards to the proposed research questions, further drawing conclusions and presenting open problems we consider warrant future work.

CHAPTER 2

Related Work

We now present the results of the literature review we have conducted, which forms the basis of knowledge upon which the remainder of our research and development work are conducted. We outline the employed methodology and describe relevant results in the literature.

2.1. Literature Review Methodology

We conducted a literature review in order to collect academically-relevant information related to our specific field of study, that is, the application of automated programming assessment to educational environments with a focus on feedback generation capabilities.

We chose to conduct an *ad hoc / snowball* approach to our literature review. Our approach was nevertheless structured and guided by several criteria used to restrict the amount of collected information to a tractable amount, with the collected publications having undergone the following general analysis:

(1) Does the publication effectively explore automated programming assessment?
(2) Does the publication explore the effect of automated feedback in programming education?
(3) Is the publication relevant in the current context of automated programming assessment systems?
(4) Does the publication explore the application of assessment systems for programming education? If so, does it contain relevant keywords, for example *white-box assessment* or *feedback generation*?

Publications subjected to the described criteria were then further analysed to determine their effective relevancy to our proposed research goals.

We additionally conducted a three-fold review step, with an initial section dedicated to motivating our work, a section dedicated to the analysis of related work into white-box assessment systems, and another focused on previous research work on how formative feedback affects programming students.

While a systematic review was not conducted, we motivate using Figure 2.1 the growing desire for a *dynamic*, *white-box*, *programming assessment* and *feedback generation* library by showing the increasing number of publications containing these keywords published per year in the ACM Digital Library.[1] We note the noticeable increase in the number of relevant publications starting around 2021, which we attribute to the research and development work conducted on automated assessment to tackle the needs introduced by

[1]The four keywords must be simultaneously contained anywhere in the publication.

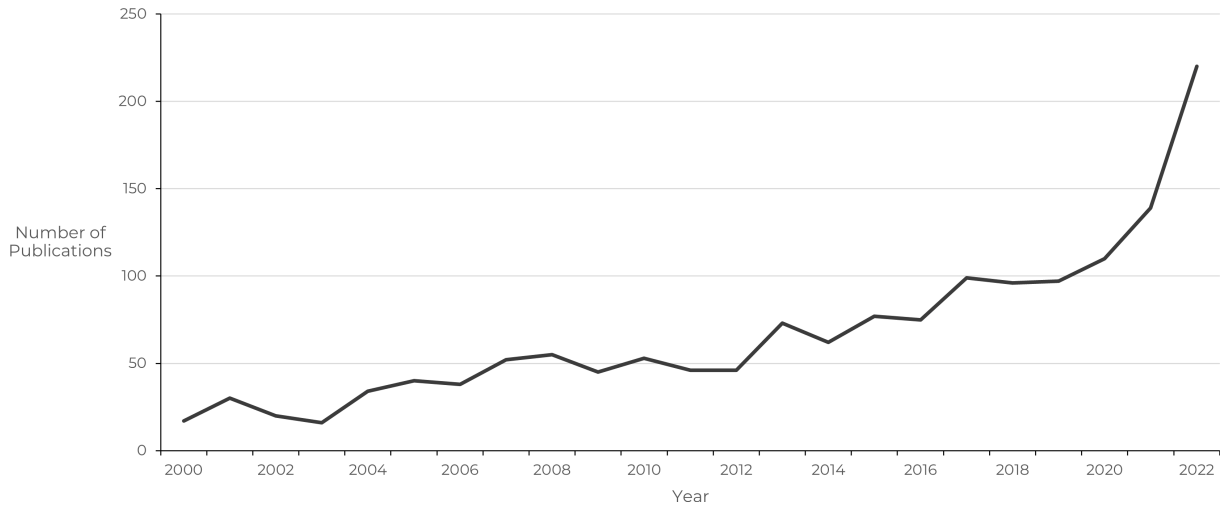

FIGURE 2.1. Number of relevant publications in the ACM Digital Library from 2000 to 2022.

TABLE 2.1. Academic publications and automated assessment tools analysed for the motivation and literature review steps. Some publications were cited in more than one Section.

Section	**Conference Papers**	**Journal Articles**	**Tools**
1.1	7	7	
2.2	7	11	6
2.3	2	4	
2.4	6	2	
	19 unique papers	**17** unique articles	**6**
	36 unique publications		

the spread of hybrid or remote teaching practices largely brought on by the COVID-19 pandemic starting in 2020.

In total, the motivation and literature review steps consisted in analysing 36 academic publications, 19 of which were conference papers and 17 were journal articles. We additionally analysed a total of 6 automated assessment and related tools for which we could not find related publications. A summary of the number of analysed tools and publications is provided in Table 2.1. The remaining Sections of this dissertation cite publications where it was deemed relevant, even if they were not included in the initial literature review.

2.2. Automated Programming Assessment

Being the most widely-known software testing platform for Java, JUnit offers extensive functionality when it comes to unit testing. While JUnit-based assessment tools can be augmented with meta-programming functionalities to offer more detailed information on program behavior, such functionality is not available out-of-the-box and requires specialized knowledge to integrate third-party libraries and/or languages. Examples include

AspectJ [**27**][2], which requires knowledge on code instrumentation, and Javassist [**10**][3], ASM [**5**][4], and ByteBuddy[5], which require knowledge of Java bytecode engineering. These fields, as described in Section 1.1, may not be well-understood by programming instructors. Similarly, Valgrind[6] is a code instrumentation toolkit that enables detection of memory management and threading bugs, along with program profiling. While these tools lay the groundwork upon which an automated assessment system may feature white-box analysis, we consider the requirement of specialised knowledge to be a negative factor when it comes to their general ease of use. Additionally, significant analysis or instrumentation is required to accurately collect a relevant amount of information during a program's execution, negatively impacting the scope for which these tools can be applied.

Lizard[7] is a multi-language static analysis tool supporting the Java language, focusing on determining the cyclomatic complexity of implemented functions, among other forms of static code analysis. While cyclomatic complexity is a useful tool for measuring and therefore managing the complexity of a program [**33**], it does not offer insight into the algorithmic behaviour of an implemented solution. Similar tools, like Cppcheck[8] for C and C++, focus on bug detection through static code analysis, providing no dynamic analysis functionalities for a program's execution.

FindBugs [**23**][9] is a static analysis tool for Java bytecode to find software defects, such as null pointer accesses, based on a catalog of bug patterns. In our approach, we are not primarily focused on finding defects, but instead on checking if the desired algorithmic behavior is met. Furthermore, our approach focuses on dynamic analysis of the code's execution as opposed to the static analysis of the compiled code.

Extensive reviews on automated assessment have been conducted, and the general findings are that most systems tend to focus on aspects other than dynamic white-box analysis [**26**, **34**, **38**]. Efforts have been made to develop software testing solutions that not only simplify the process on the instructor's side but also collect more detailed metrics regarding the evaluated code to allow the elaboration of more detailed feedback. Most of these, however, introduce dependencies on either static code analysis, instrumentation, or a combination of both techniques [**12**, **26**]. While these tools lay the groundwork upon which an automated assessment system can be implemented in such a way that detailed white-box information can be collected, we consider the requirement of specialised knowledge to be a negative factor when it comes to the tools' ease of use to general instructors or practitioners.

[2] `https://www.eclipse.org/aspectj`
[3] `https://www.javassist.org`
[4] `https://asm.ow2.io`
[5] `https://bytebuddy.net/`
[6] `https://valgrind.org/`
[7] `https://github.com/terryyin/lizard`
[8] `https://cppcheck.sourceforge.io/`
[9] `https://findbugs.sourceforge.net`

JavAssess [**24**] is a Java library used to integrate deeper code analysis capabilities into existing automated assessment tools. Their approach relies on combining traditional black-box unit testing with code instrumentation and meta-programming functionalities, yet does not offer a way to dynamically collect white-box execution metrics. Nonetheless, the goals of JavAssess are considerably similar to Witter's, aiming to be a library that can be integrated into existing automated assessment systems to facilitate a deeper analysis of student code, and the two libraries could work in tandem to provide a comprehensive assessment toolkit. AutoGrader [**21**] is a similar assessment library, leveraging on the meta-programming functionalities of the Java language along with typical unit testing for code assessment. Their approach, again, does not contemplate the dynamic collection of white-box metrics describing code execution.

Pedal [**19**] is an infrastructure, supporting the Python language, for automatic code evaluation and feedback generation that tackles problems similar to those we address, namely the importance of an in-depth analysis of student code for the generation of detailed feedback. Their approach relies on static code analysis along with traditional unit test execution. While the usage of static code analysis allows for feedback much more detailed than simple unit testing systems, it does not support measuring dynamic, internal details of a program's execution, which is Witter's main focus.

GitHub Classroom[10] is an educational system built on GitHub which allows educators to automate the assignment submission and evaluation process for students. Their assignment testing capabilities rely on comparing outputs against reference solutions and checking whether the execution of test commands results in failure[11].

ConGu [**13**] is a runtime verification tool that enables the assessment of object-oriented solutions against formal algebraic specifications. Its main goal is to test abstract data types against function domain restrictions and algebraic conditions or axioms that function applications must verify.

Jeed[12] is a toolkit for Java and Kotlin in-memory execution with a focus on safety and performance. While Jeed's goals align with those of our proposed library by enabling code execution in a sandboxed environment providing access to code evaluation metrics, the assessment is focused on source code analysis rather than dynamic runtime events. Namely, Jeed supports linting, cyclomatic complexity analysis, and a listing of which language features are present in a given program.

Mooshak [**29**] is an online programming assessment tool that checks whether a submitted program functions correctly. To this effect, Mooshak analyses the programs for their returned or printed outputs, and if any compilation or runtime errors were produced [**43**]. Mooshak's goal is broad, aiming to be a full online programming context judge for several programming languages [**29**, **43**], and as such provides functionalities to execute custom

[10] `https://classroom.github.com/`
[11] `https://docs.github.com/en/education/manage-coursework-with-github-classroom/teach-with-github-classroom/use-autograding`
[12] `https://github.com/cs125-illinois/jeed`

static and dynamic analysers for white-box assessment. This aligns with our goal of providing Witter as a library which is easily integrable into existing assessment systems.

Several Intelligent Tutoring System (ITS) have been proposed or developed for Java, supporting feedback generation systems whose goals align with those of our work. An ITS is a system aiming to emulate a human instructor to provide students with personalised instruction [**48**].

Java Sensei [**48**] is an ITS for the Java language aiming to fully simulate the function of a human instructor in a programming class by implementing, among other components, an agent that displays facial expressions as a form of affective feedback for the student.

Similarly, JITS [**42**], an ITS developed for the Java language, aims to provide students with an interactive and intuitive learning environment for Java. Interestingly, the focus of the authors seems to be on aiding students with the process of correcting syntax errors, and an error correction algorithm was developed to this effect.

Further examples of ITS for the Java language include FIT [**18**] and J-LATTE [**22**]. FIT provides students with a development environment where automated feedback on the current implementation can be requested, and where progress towards a correct solution can be tracked through the displayed correctness indicators [**18**]. J-LATTE provides students with a structured editor, helping to familiarise students with the syntactic structure of the Java language. Assignments are evaluated by J-LATTE by checking that the student's code verifies a given set of constraints [**22**]. The goals of these systems align with those of our approach in the sense that they also prioritise student autonomy and creativity, as opposed to restricting students' implementations.

While the overall goal of an ITS is much broader than what we consider in our approach, aiming to provide a full development environment simulating the tasks of a personalised instructor for each student, the automated assessment and feedback step could be augmented with the testing capabilities of our proposed library.

While the described approaches and tools provide ample research and development opportunities, we now explore some shortcomings which can be identified in existing automated assessment systems when it comes to the technical aspects of their implementation and the quality and effects of the feedback they provide to programming students.

2.3. Shortcomings of Automated Assessment

Shortcomings can be identified in the two perspectives related to automated programming assessment - that of the instructor, who has to define the test cases, and that of the student, whose implementation is subjected to the assessment.

As described in Section 1.1, most automated assessment systems can be augmented beyond black-box testing by means of specialised languages features such as code instrumentation or meta-programming, requiring instructors to be proficient in these domains, as noted in Paiva et al. [**38**] and Messer et al. [**34**]. Moreover, in Vytautas et al. [**15**], the authors note that the specific knowledge required to utilise these features leads to them being unfamiliar to most practitioners that do not specialise in their use.

Programming instructors additionally describe several challenges pertaining to introductory programming courses. Namely, instructors state a desire for assessment systems that allow for further in-class activities to be conducted that guarantee students actively pursue solutions to their proposed assignments, but that available systems do not allow for such practices, as found in Mirhosseini et al. [**35**]. In their study, it is further highlighted that programming instructors recognise the relevancy and difficulty of utilising assessment tools that aid in student grading and additionally provide high-quality feedback that positively contributes to a student's learning process [**35**]. It may be of interest to note that some instructors are strongly opposed to automated assessment, as they consider the usefulness of feedback given to students to stem from the personal aspect of manual grading [**35**]. In our opinion, this is not an argument against automated assessment but a motivating argument towards automated assessment systems that effectively provide high-quality feedback for students.

As for students' perspectives, in Simon et al. [**8**] the authors find that students of introductory-level programming courses highlight concerns regarding the clarity of an assignment's requirements and submission process, and with the quality of the feedback they receive for their implementations, whether or not they're marked as correct. We take this as further motivation for the implementation of an assessment tool that guides students during their implementation and provides them with high-quality feedback regarding their code.

Moreover, the implementation of automated assessment tools with perceived sub-par functionalities can cause dissatisfaction and negatively impact students. In Sánchez et al. [**43**], the authors studied the effect of implementing Mooshak [**29**][13], an automated programming assessment and feedback generation tool, in an introductory programming course. Their research concluded that students expressed significant dissatisfaction with the quality of the generated feedback, which seemed to be correlated to an increased workload by the students [**43**]. We take this as further motivation towards the refinement of the feedback generated by our proposed library.

2.4. Impact of Feedback in Programming Education

It is well-accepted that timely, high-quality and effective feedback is instrumental in improving the learning progress of students [**35**, **46**]. In general, the learning process of students in any field can be improved by the introduction of formative feedback, so long as it is implemented in a manner effective for each particular field [**20**].

In Fabienne et al. [**45**], a study was conducted regarding the implementation of computer-based assessment in higher education, though not specifically for programming education. Their analysis finds that students tend to pay more attention to instantaneous or real-time feedback as opposed to delayed feedback, and that students perceived a positive impact of the availability of instant, detailed feedback on their responses.

[13] `https://mooshak.dcc.fc.up.pt/`

In Lubarda et al. [**32**], the authors explore the effects on students of automated instantaneous feedback, among other aspects, in an introductory-level programming course for engineers. Their study concludes that real-time feedback on assignments was perceived positively by students in regards to its effect on their learning process, having aided then in self-assessing their learning process and providing further motivation to improve their performance during the course. Similar findings were reported in Jansen et al. [**25**] and van den Aker et al. [**44**].

Similarly, in Venables et al. [**46**], it was found that students were more willing to self-assess and autonomously improve their own work if they had access to a tool that could provide instant feedback for their submissions.

Despite their possible benefits, concerns can be raised that students tend to over-rely on the feedback generated by the automated assessment tools, negatively impacting their autonomy in the development process. In Mitra et al. [**36**], the authors voiced this particular concern and studied the impact of an automated feedback generation tool from this perspective. In particular, the authors studied whether a tool for automated assessment and feedback generation discouraged students from autonomously testing their implementations, and if the tool increased the correctness of the students' implementations. Regarding the first concern, their study finds that the introduction of a feedback generation tools did not seem to hinder students in their acquisition of autonomous software testing skills. Additionally, their obtained results indicate that the usage of the tool significantly improved the correctness of students' implementations.

Another possible benefit pertains to student engagement. In Benotti et al. [**3**], the authors found that the usage of a web-based automated assessment tool[14] in a programming class significantly reduced student dropout rates when compared to a semester in which the same course was conducted without the usage of any automated assessment tools.

One of our main goals with our proposed library is to provide a way to collect information regarding students' implementations which can be used to not only guide and improve their learning and self-assessment process, but also to aid instructors in providing students with higher-quality, formative feedback and in structuring programming assignments in such a way that benefits students' learning process.

[14] `https://mumuki.io/`

CHAPTER 3

Background

In this chapter, we present the underlying concepts and libraries upon which Witter is built. We offer an overview of the libraries' concepts and functionality, which form the basis for the more comprehensive description of our work presented in Chapter 4.

3.1. JavaParser Library

Abstract Syntax Trees (ASTs) are data structures which represent the structure of a given program or piece of source code written in a programming language. We say that the syntactic structure represented by these structures is *abstract* in the sense that non-structural syntax details are omitted. For example, in Java, the body of a method must be enclosed in curly brackets and each statement must end in a semicolon. In an abstract syntactic representation, we only consider that a method has a signature and a body consisting of a list of statements, and it is thus unnecessary to represent details like brackets or semicolons. The term *tree* stems from the fact that each program construct is represented with a node, and links are added between each node to represent the structure of the program.

A possible application of ASTs lies in executing programs through *abstract syntax tree interpretation*, an approach where an Abstract Syntax Tree is constructed from the program's source code, and an interpreter then executes this program by traversing the structure of the tree and executing the logic associated with each node.

Abstract Syntax Trees can also be used for translating source code between two programming languages, with the abstraction offered by the AST serving as an intermediate representation which facilitates the translation of each language construct by enabling a mapping between individual constructs as they are encountered when traversing the tree. It is this principle that the Strudel library employs in producing its model of structured programming languages, which is analysed in further detail in Section 3.2.

JavaParser[1] is an open-source Java library implementing a parser for Java source code. The parser produces an AST of Java code which can be manipulated to analyse or instrument existing code or programmatically generate new code. As a simple illustrative example, Figure 3.1b shows the AST produced by the JavaParser library for the source code present in Figure 3.1a. Note that, as described, syntactic details like delimiters or grouping parentheses are not explicitly represented in the AST, which only describes the source code's structural constructs. Further note that the structure of each node contains additional information regarding the program, such as specifying the method's return type

[1] `https://javaparser.org/`

(A) Source Code.

```
int bar() {
    return 42;
}
```

(B) Abstract Syntax Tree.

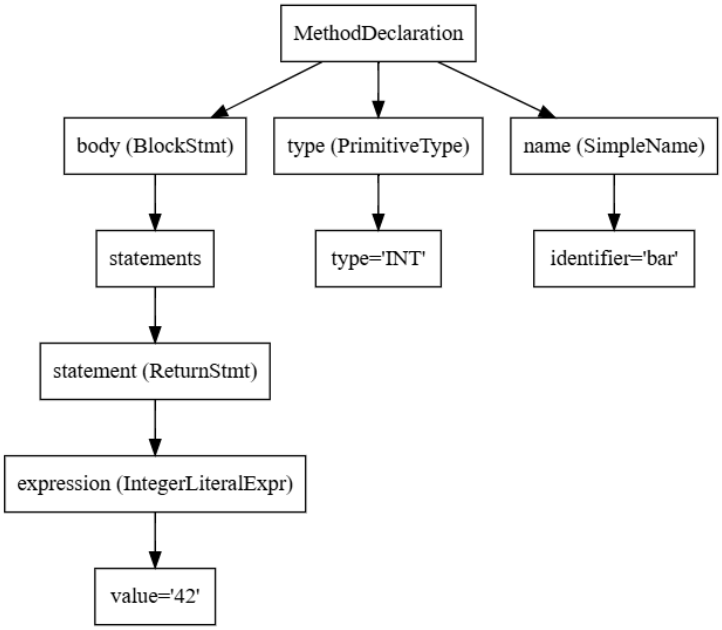

FIGURE 3.1. Example of an AST for the Java language.

as a primitive type identified with the code `INT` (integer), or that the return value is given by an integer literal expression with value 42.

We utilise the JavaParser library to parse and generate Abstract Syntax Trees from Java source code to implement core features of both the Strudel and Witter libraries. Implementation details of Witter's usage of JavaParser are described in Section 4.3.

3.2. Strudel Library

Strudel lies at the core of Witter, providing the environment which makes its functionalities possible. While the original work on Strudel was independent from Witter's development and this dissertation, the importance of the library as a base for Witter and the contributions brought on by the development of our current work make it relevant to describe the library in further detail.

The general workflow of using Strudel is illustrated in Figure 3.2. We begin by describing the core concepts and motivating Strudel's usefulness as an execution environment for Witter in Section 3.2.1. The process of instantiating program models which can be executed in Strudel is described in further detail in Section 3.2.2, with a special focus being given to the parsing and translation of Java source code, where the contributions from the work described in this dissertation were most extensive. Finally, in Section 3.2.3, we provide examples of utilising the Strudel library to capture execution information about a program to illustrate how this information is used within the Witter library.

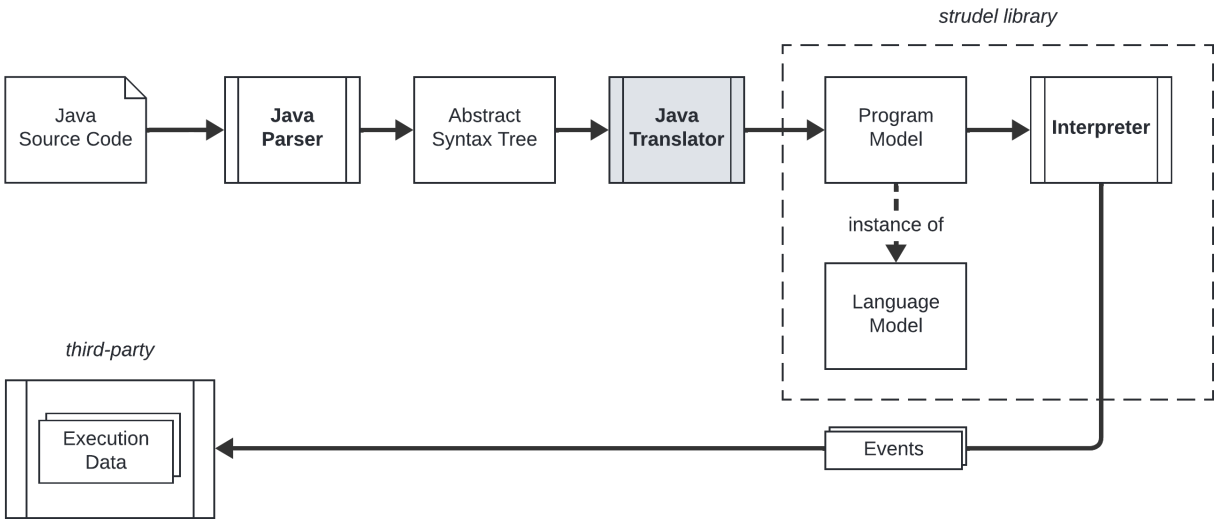

FIGURE 3.2. Workflow of the Strudel library.

3.2.1. Introduction

Strudel is a programming library providing a set of constructs which model concepts of structured programming, which can be instantiated to produce models of programs. These models can be used for static analysis, or interpreted to simulate the program's execution.

Strudel's internal model provides the necessary constructs for modelling programs following a structured programming paradigm. The core concepts are described in Figure 3.3, which illustrates the implemented abstractions for modelling control structures along with procedure definition and invocation.

The hierarchy defined by these abstractions is similar to the usual hierarchy in object-oriented programming. A *module* holds a set of *procedures* which hold a set of *variables*, have a return *type*, and whose bodies constitute *blocks*. These, in turn, hold a set of *statements*, other blocks, or *control structures*. The latter can constitute selections (`if-else`) or loops (`while`), both of which which hold a *guard* (condition determining whether the block should execute) and, in the case of selections, an optional alternative block (`else`) which executes if the guard fails.

The `return`, `break`, and `continue` statements function in the same manner as their Java counterparts, respectively terminating a procedure's execution, terminating a loop's execution, and skipping a loop to its next iteration.

Strudel provides a virtual machine that executes these abstractions as programs in a call stack-based environment. The *state* of a program is thus held in *heap memory* and a *call stack*. The heap holds chunks of memory which correspond either to *arrays* or *records* (object instances). The call stack holds the *stack frames* associated with each procedure call, each of which contains the values associated with each variable within the procedure.

The goal of the Strudel library is to provide an environment for fine-grained observation of code execution events. As such, the virtual machine exposes every aspect of a program's execution, allowing listeners to be injected into the execution process to capture information.

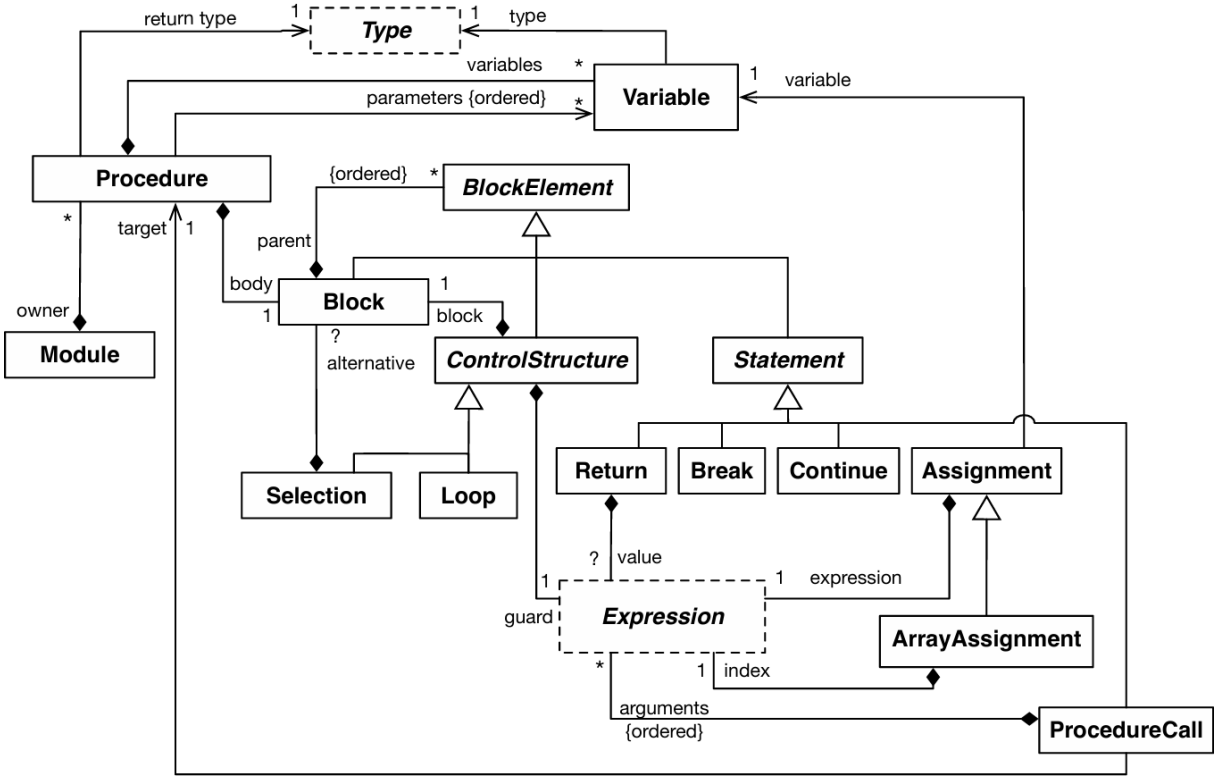

FIGURE 3.3. Strudel model for control-flow. Names in *italics* denote abstract concepts.

Namely, Strudel implements a Listener system which currently supports the notification of the following events:

- Procedure calls and terminations, along with `return` calls;
- Variable, array element, and field assignments;
- Array and object allocations;
- Array element and object field changes;
- Loop iterations and terminations;
- Expression evaluations;
- Execution errors;
- System output.

This makes Strudel uniquely suited to Witter's goal of providing a program assessment tool based on code execution, with Witter thus taking full advantage of Strudel's virtual machine for running and capturing information regarding student code. The presence of a fine-tuned execution environment tailored to Witter's goals offers a seamless way to implement our desired features, along with providing fine-tuning and improvement opportunities that would otherwise be intractable to implement using techniques such as bytecode instrumentation.

3.2.2. Java Translation

Strudel provides an internal Domain-Specific Language (DSL) that enables the manual instantiation of the classes which make up its internal model of structured programming.

However, in order to provide an accessible and practical way of creating program models, the library implements a translator supporting a subset of Java language features, and which creates models from Java source code. The translator was implemented using the JavaParser library, which builds an AST from the provided Java source code that can then be translated to Strudel's internal model.

The translation process is straightforward and is achieved incrementally as the AST generated from the Java source code is traversed. This is achieved by implementing procedures which provide mappings between the different types of Java statements and expressions to their corresponding representations in Strudel's internal model. For example, the `return`, `break`, and `continue` Java statements are translated to the `Return`, `Break`, and `Continue` Strudel abstractions, respectively, as illustrated in Figure 3.3.

Firstly, the translator is provided with either raw Java source code or a reference to a file containing the source code. A Strudel `Record` type is created to represent each class contained in the source code. Each field of the class declared in the source code is then translated to Strudel's internal model and stored within the corresponding record type. The type of each field must be translated accordingly, and field initialisers are supported by injecting each initialisation expression into their declaring record's constructors.

The translation of a class's declared procedures is achieved by translating its return type and parameters, along with sequentially translating each statement in a procedure's body, and aggregating the translated statements within a procedure represented in Strudel's internal model. Currently, translation is implemented for the following statement types:

- The `return` and `throw` statements;
- Conditional statements (`if-else`);
- Loop statements (`while`, `for`, `for-each`), along with the `break` and `continue` statements;
- Expression statements (e.g. procedure calls, object creation).

As there can be several expressions contained within statements or within other expressions, their translation is achieved recursively. For example, the translation of a variable assignment expression `x = y + 3` breaks down into:

(1) The translation of the left-hand-side as a variable target expression;
(2) The translation of the right-hand-side as a binary expression which applies the summation operator to a variable reference expression and an integer literal expression.

Again, this process is achieved in a simplified manner by implementing a recursive procedure mapping each Java expression type to its corresponding Strudel abstraction. Currently, the following expression types are supported:

- Variable, array, and object field assignments;
- Unary expressions (e.g. `i++`);
- Binary expressions (e.g. `i + j`)
- Method call expressions;

- Literal expressions for primitive types (integer, double, boolean, and character), along with strings and `null`;
- Array creation, initialisation, and access expressions;
- Object creation and field access expressions;
- Conditional expressions (e.g. Java's ternary operator, `condition ? if : else`).

Initially, the scope of Strudel's supported Java translation features was limited to those necessary to conform to the test cases then defined. Witter's development revealed opportunities to augment Strudel's translation to broaden the scope of supported assignments, and contributions were thus made for the translation of field initialisers, foreign Java procedure calls (e.g. `System.out.println`), instantiation of foreign objects (e.g. `new java.util.Scanner()`), polymorphic Java types (i.e. interfaces), and nested types. Minor contributions were made for the translation of variable declaration statements containing multiple variables, the usage of unary increment and decrement operators (i.e. `++` and `-`) within array accesses (e.g. `a[i++]`), and the usage of generic type parameters.

It is relevant to note that the translation process is independent from the interpretation of Strudel's internal model, which does not depend on an external language. Strudel itself is a language-agnostic interpreter, and the current translation module simply enables the construction of Strudel program models from Java source code, demonstrating Strudel's applicability to any adequate programming language and providing a way of running experiments with Java assignments, enabling us to explore Research Question **RQ2**. The extension of Strudel to other structured programming languages should only require the implementation of a module which translates source code in the target language to an equivalent representation in Strudel's internal language model.

3.2.3. Examples

We now provide examples of how Strudel can be used to execute Java source code and to obtain information regarding the code's execution.

Consider an example where, after executing a given procedure, we want to know how many total array positions were allocated. To this effect, we pass the code present in Figure 3.4a to a Strudel virtual machine unto which we attach a listener that tracks array allocation events to accumulate the number of allocated positions, as illustrated in figure 3.4b. For simplicity, after the procedure executes, we simply print this number to the console. The listener attached to the virtual machine being used increments an accumulator variable with an array's length every time one is allocated. As such, the allocation of the two arrays within the considered procedure, of lengths 10 and 20, respectively, lead to the value 30 being printed in the console when the procedure finishes executing.

Consider now that we want to count the number of loop iterations for a given procedure. The process for achieving this is similar to the previous example, as we just need to specify a different listener event to gather the relevant information. For an invocation with the value 10 passed as the procedure's argument, we expect the total number of iterations to be 10, and this is the value printed to the console. We note the possibility of using

similar methods to construct, for example, a model for the number of loop iterations as a function of the input size, providing analysis opportunities relevant to fields such as algorithm analysis.

(A) Java source code (SourceCode.java).

```
static void foo() {
    int[] a = new int[10];
    int[] b = new int[20];
    a[0] = 5;
    b[0] = a[0];
}
```

(B) Executing Java source code and listening for array allocations.

```
val vm = IVirtualMachine.create()
val model = Java2Strudel().load(File("path/to/SourceCode.java"))
var allocatedLength = 0
vm.addListener(object : IVirtualMachine.IListener {
    override fun arrayAllocated(ref: IReference<IArray>) {
        allocatedLength += ref.target.length
    }
})
vm.execute(model.getProcedure("foo"))
println("Total allocated array positions: $allocatedLength")
```

(C) Console output.

```
Total allocated array positions: 30
```

FIGURE 3.4. Example: Tracking array allocations in Strudel.

(A) Java source code (SourceCode.java).

```
static int bar(int n) {
    int i = 0;
    for (int j = 1; j <= n; j++) i += j;
    return i;
}
```

(B) Executing Java source code and listening for loop iterations.

```
val vm = IVirtualMachine.create()
val model = Java2Strudel().load(File("path/to/SourceCode.java"))
var iterations = 0
vm.addListener(object : IVirtualMachine.IListener {
    override fun loopIteration(loop: ILoop) { iterations++ }
})
vm.execute(model.getProcedure("bar"), 10)
println("Total loop iterations: $iterations")
```

(C) Console output.

```
Total loop iterations: 10
```

FIGURE 3.5. Example: Tracking loop iterations in Strudel.

CHAPTER 4

Witter Library

We explore research question **RQ1** by implementing Witter, a software library which aims at providing a testing tool for introductory programming exercises in Java. In this Chapter, we describe the library's structure and how it can be used to assess programming exercises. We finish by discussing the technical aspects of the library's implementation.

4.1. Overview

In Witter, instructors provide reference code solutions to assignments, which constitute the *reference solutions* used to evaluate students' code. Instructors can define test cases based on these reference solutions through two alternative methods:

(1) By annotating the reference solution directly using Witter's *test specification language*, as described in Section 4.2.1.
(2) By using Witter's *domain-specific language* (DSL) to define test cases programmatically, as described in Section 4.2.2.

Students, in turn, simply need to submit solutions that are compared against the reference solutions using *runtime metrics*, such as the measuring of executed operations or allocated memory. The general workflow for using Witter is illustrated in Figure 4.1.

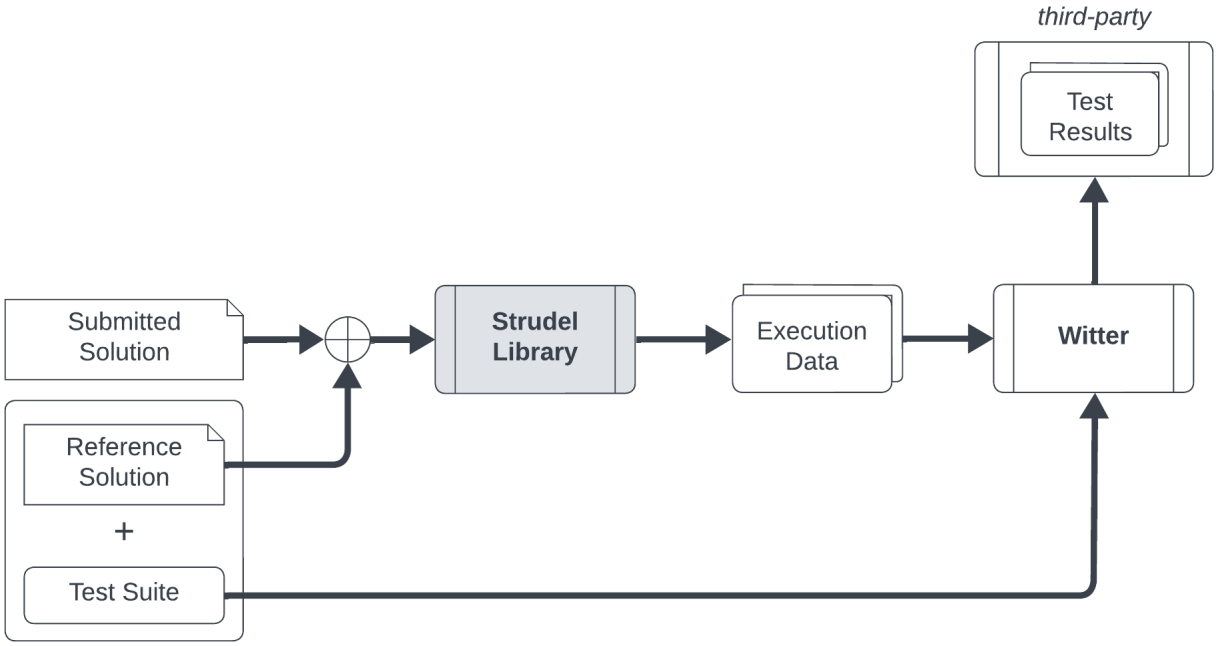

FIGURE 4.1. Workflow of the Witter library. Components in grey are dependencies from external libraries (Strudel).

Table 4.1 presents the set of metrics we currently support. Note that outputs of functions are also measured, to allow for regular black-box testing, while all the other items are mostly useful for white-box testing. As Witter runs on Strudel, an execution environment that supports the dynamic collection of the described metrics or their constituent parts, no static analysis is needed for these measurements. The optional parameter *threshold* that some annotations have is useful to tolerate slight deviations from the reference solution (most often ± 1). If not specified, the *threshold* value is zero.

As Witter is designed for third-party integration, we provide a form of executing the tests programmatically. Tests are executed either by providing an annotated reference solution or a test suite defined using the DSL, and a solution that one wishes to assess. The specific method of running tests defined either using annotated code solutions or through Witter's DSL are described in Sections 4.2.1 and 4.2.2, respectively.

TABLE 4.1. Runtime metrics and corresponding test specification annotations.

Metric	**Annotation**	**Verification**
Return values	`@Test(`*[...args]*`)`	Return value is equal to reference solution. Multiple annotations can be used.
Side effects	`@CheckSideEffects`	Side effects on arguments (presence and absence) are the same to those of the reference solution.
Loop iterations	`@CountLoopIterations(`*[threshold]*`)`	Total number of loop iterations matches the one of the reference solution.
Array allocations	`@CheckArrayAllocations`	The array allocations match those of the reference solution (component types and lengths).
Array reads	`@CountArrayReads(`*[threshold]*`)`	The number of array read accesses is the same as in the reference solution.
Array writes	`@CountArrayWrites(`*[threshold]*`)`	The number of array write accesses is the same as in the reference solution.
Object allocations	`@CheckObjectAllocations`	The number of object allocations and their types match those of the reference solution.
Recursive calls	`@CountRecursiveCalls(`*[threshold]*`)`	The number of recursive calls matches the one of the reference solution.

The test results produced by Witter consist of a list of feedback items for each aspect defined in the test specification, holding the following information:

- A flag indicating success or failure;
- Which kind of metric has failed (recall Table 4.1);
- The location of code elements involved in the failed tests (e.g., procedure, parameters, loop structures);
- A human-readable descriptive feedback message.

The implementation of two alternative test specification methods is due to the fact that many typical introductory programming assignments require the implementation of object-oriented solutions such as data structures. Appropriate testing for this kind of exercise would require tests that perform a sequence of several assertions over the same object or interaction between different objects. Witter's DSL is more appropriate for this kind of problem, which would be considerably difficult to reliably tackle purely using annotated code solutions. For simple static procedure testing, the process of defining tests using the DSL can be avoided by simply annotating a reference solution.

4.2. Test Specification

4.2.1. Annotated Code Solutions

One can define the test cases for a given exercise by writing a reference solution in a Java method, annotated with a header comment that defines the different test inputs and the metrics that should be measured during test execution. The content of the comments has to obey Witter's code annotation syntax, which is similar to Java's annotation syntax. The test specification relying on purely textual comments, as opposed to adopting regular annotations, is intended to simplify the process by not requiring external annotation types, while also giving more freedom with respect to the type of content in the annotations.

Figure 4.2a illustrates an example of specifying a Witter test for a simple function that calculates the sum of an array of integers. The `@Test` annotation serves the purpose of a black-box test case, containing the arguments that should be passed when executing the method for a single test case. The arguments are enclosed in parentheses and follow Java's usual syntax. When comparing solutions, all the defined runtime metrics will be checked independently for each of these test cases.

We now present examples of how Witter's white-box testing functionalities using annotated code solutions could be used to assess typical introductory programming assignments. For each example, we present a reference solution, a hypothetical solution to test, and Witter's test result output when doing a console-based usage. Note that if using Witter programmatically, one may inspect the feedback details in isolation and obtain more information than what is presented here.

Factorial (Recursive). Consider the classical example of a recursive implementation of the factorial function as an exercise (see Figure 4.3a). In this case, the test specification

(A) Test specification example.

```
/*
@Test({1, 2, 3, 4, 5})
@Test({2, 4, 6})
@CountLoopIterations
@CountArrayReads
@CheckSideEffects
*/
public static int sum(int[] a) {
    int result = 0;
    for (int i = 0; i < a.length; i++) {
        result = result + a[i];
    }
    return result;
}
```

(B) Testing an arbitrary solution against a reference solution.

```
Test test = new Test("ReferenceSolution.java");
List<TestResult> results = test.execute("Solution.java");
```

FIGURE 4.2. Witter API for test specification and execution using annotated code solutions.

only has a single test case (`@Test`), and we impose the restriction that the solution must be implemented recursively `@CountRecursiveCalls`, tolerating a deviation of one call.

We present a hypothetical incorrect solution (see Figure 4.3b), where not only was the algorithm implemented iteratively, but the iteration was defined to start at an incorrect value, leading to an incorrect result.

In this example, we simultaneously demonstrate Witter's both black and white-box functionalities. Figure 4.3c presents the test output, where one can see that both aspects of the same test case fail. As in tools like JUnit, we present the expected values and the ones that were found.

Binary Search (Iterative). Consider an exercise for implementing a binary search over an array of integers that returns the array index where the number is stored, or -1 otherwise (see Figure 4.4a). The test specification defines two test cases, one positive and one negative. In contrast to the previous example, here we aim at an iterative implementation, and hence, we check if the number of loop iterations matches (`@CountLoopIterations`). We also check if the implementation has no side effects (`@CheckSideEffects`).

We present a hypothetical incorrect solution (see Figure 4.4b), which despite producing the same result (black-box test), is performing a linear search. Therefore, in this sort of situation, as the incorrectness is not as easily noticeable as when a functional test fails, a student could easily proceed to the next exercise given that the expected result matches.

Even though the test results indicate that both cases have the expected result (see Figure 4.4c), they both fail with respect to the number of expected loop iterations. When

no side effects are expected, and the solution under testing also has no side effects, Witter does not report a successful test. A failing test would be given only in case of a mismatch.

Insertion Sort (Procedure). Consider an exercise to implement the insertion sorting algorithm as a procedure that modifies an array of integers (see Figure 4.5a). Here we want to check that the side effects (array becomes sorted) match those of the reference solution (`@CheckSideEffects`). As there are several sorting algorithms the student could implement, in this case, we aim at checking that the behavior of the solution under testing actually performs insertion sorting. We achieve this by counting the number of array accesses (`@CountArrayReads` and `@CountArrayWrites`).

We present a hypothetical incorrect solution (see Figure 4.5b), which performs a correct sorting, but through the selection sorting algorithm. As with the previous example, a student providing such a working solution may miss the point of the exercise if only black-box tests are performed.

The test output will indicate that the sorting result is correct (expected side effects), but it will also report the mismatches in both array reads and writes, an indication that the intended algorithm implementation is not correct.

(A) Reference solution with recursion.

```
/*
@Test(5)
@CountRecursiveCalls(1)
*/
static int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n - 1);
}
```

(B) Solution under testing (iterative, with a defect).

```
static int factorial(int n) {
    int f = 1;
    for (int i = 0; i <= n; i++)
        f *= i; // i starts at 0, f always 0
    return f;
}
```

(C) Witter test results (black-box and white-box fail).

```
[fail] factorial(5)
       Expected result: 120
       Found: 0

[fail] factorial(5)
       Expected: 4 recursive calls (± 1)
       Found: 0
```

FIGURE 4.3. Example: factorial testing using an annotated solution in Witter.

(A) Reference solution using binary search.

```
/*
@Test({1, 2, 3, 4, 5, 6, 7}, 1)
@Test({1, 3, 7, 9, 11, 13, 17, 19}, 18)
@CountLoopIterations
@CheckSideEffects
*/
static int binarySearch(int[] a, int e) {
    int l = 0;
    int r = a.length - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (a[m] == e) return m;
        if (a[m] < e) l = m + 1;
        else r = m - 1;
    }
    return -1;
}
```

(B) Solution under testing (performing linear search).

```
static int binarySearch(int[] a, int e) {
    for (int i = 0; i < a.length; i++)
        if (a[i] == e) return i;
    return -1;
}
```

(C) Witter test results (black-box pass, white-box fail).

```
[pass] search([1, 2, 3, 4, 5, 6, 7], 1)
       Expected result: 0

[fail] search([1, 2, 3, 4, 5, 6, 7], 1)
       Expected: 3 loop iterations (± 0)
       Found: 1

[pass] search([1, 3, 7, 9, 11, 13, 17, 19], 18)
       Expected result: -1

[fail] search([1, 3, 7, 9, 11, 13, 17, 19], 18)
       Expected: 4 loop iterations (± 0)
       Found: 8
```

FIGURE 4.4. Example: binary search testing using an annotated solution in Witter.

4.2.2. Domain-Specific Language

In order to tackle the inherent difficulty of supporting the assessment of object-oriented implementations through annotated code solutions, we implemented an internal DSL in Kotlin providing a programmatic way for defining stateful test cases. We consider that a simple DSL requires less effort from instructors for defining test cases for student assignments, as the only required knowledge is that of the DSL's syntax, as opposed to knowledge on specialised topics like code instrumentation or meta-programming.

(A) Reference solution performing insertion sort.

```
/*
@Test({5, 4, 3, 2, 1})
@CountArrayReads
@CountArrayWrites
@CheckSideEffects
*/
static void sort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j] >= a[j - 1]) break;
            int tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
        }
    }
}
```

(B) Solution under testing (performing selection sort).

```
static void sort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < a.length; j++)
            if (a[j] < a[min]) min = j;
        int tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

(C) Witter test results (black-box pass, white-box fail)

```
[pass] sort({5, 4, 3, 2, 1})
       Expected side effects: {1, 2, 3, 4, 5}

[fail] sort({5, 4, 3, 2, 1})
       Expected array reads: 40
       Found: 28

[fail] sort({5, 4, 3, 2, 1})
       Expected array writes: 20
       Found: 8
```

FIGURE 4.5. Example: insertion sort testing using an annotated solution in Witter.

Figure 4.6 illustrates Witter's DSL with a *test suite* for list data structures, containing two *test cases*. These test cases can be configured to use any number of white-box metrics either throughout the test or within a bounded scope (*using* directive). As in the initial version of Witter, the supported evaluation metrics, whose usage within the DSL is summarised in Table 4.2, can be optionally instantiated with a *margin* parameter that specifies an acceptable deviation interval from the reference value, in order not to constrain students' code to a single, rigid solution.

TABLE 4.2. DSL runtime metrics.

Metric	Usage
Loop Iterations	`CountLoopIterations(`*[margin]*`)`
Array Reads	`CountArrayReadAccesses(`*[margin]*`)`
Array Writes	`CountArrayWriteAccesses(`*[margin]*`)`
Recursive Calls	`CountRecursiveCalls(`*[margin]*`)`
Object Allocations	`CheckObjectAllocations()`
Array Allocations	`CheckArrayAllocations()`
Side Effects	`CheckSideEffects()`

```
val tests = TestSuite(referencePath = "path/reference/List.java") {
    Case("testContains") {
        // Create new object and store a reference to it
        val list = ref { new("List") }

        // Executed without white-box metrics (black-box only)
        list.call("size") // 0
        list.call("add", "hello")
        list.call("size") // 1
        list.call("add", "world")
        list.call("size") // 2

        using(CountLoopIterations() + CountArrayReadAccesses()) {
            // These calls compare loop iterations
            list.call("contains", "hello") // true
            list.call("contains", "algorithm") // false
        }
    }

    // All the calls within this case compare loop iterations
    Case(CountLoopIterations(), "testIsEmpty") {
        val list = ref { new("List") }
        list.call("isEmpty", expected = true)
        list.call("add", "hello")
        list.call("isEmpty", expected = false)
    }
}
```

FIGURE 4.6. Witter's DSL syntax. Example Test Suite for a list data structure comprising the operations *add*, *contains* and *isEmpty*.

An object can be created using the *new* directive by passing the name of the class to instantiate followed by a list of arguments to one of the class constructors. References to the created objects can be stored using *ref*. Class methods can be invoked by using the *call* directive on a previously declared reference. A sequence of these directives defines a stateful test case.

The *call* directive is used by specifying the name of the method to be invoked and a list of arguments. We may use the "dot notation" to perform calls on instance methods given its reference (`ref.call(...)`), as opposed to calling an instance method by passing

its reference as the first argument (`call("methodName", ref, ...)`). For every call, the return values of the evaluated method are compared to the reference solution, allowing for regular black-box testing. Additionally, if the optional *expected* argument is passed, Witter will assert that both the reference solution and the solution under evaluation produce the expected result. This verification allows educators to assert that their solution works as expected, preventing the accidental usage of a faulty reference solution as the ground truth for evaluating students' implementations.

An automated assessment system may import Witter as a third-party library. In this example we are merely displaying the objects of the test results to the console, but these can be inspected for custom reporting, depending on the assessment system.

We now present examples of how Witter's DSL can be used to test typical introductory programming assignments. We present a reference solution for each example, along with hypothetical solutions under testing, the tests defined using the DSL, and Witter's output in a console-based environment.

Array Average (Procedure). Consider an assignment where a student must implement a function for calculating the average of an array of double values (see Figure 4.7a). We wish to assess not only the correctness of the produced result, but also that of the algorithm behaviour by checking that the number of loop iterations matches that of the reference solution (`CountLoopIterations`). We thus apply the tests defined using Witter's DSL, as illustrated in Figure 4.7b, to a hypothetical incorrect solution which does not consider the first element of the array (see Figure 4.7c). We can see from the results in Figure 4.7e that white-box testing is especially useful in cases where the results produced by an implementation are correct despite incorrect algorithm behaviour.

Stack (Data Structure). We now illustrate Witter's data structure procedure testing functionalities using the presented DSL. Consider an assignment given to students which specifies they must implement a Stack data structure. We impose the restriction of the *size* method, which counts the number of items in the array, executing in constant time and requiring no array access operations. For simplicity, we consider an array of integers. The ideal solution is to add an integer attribute to the Stack type which tracks the number of elements as they are added to and removed from the data structure, as illustrated in Figure 4.8a. We present in Figure 4.8c a hypothetical incorrect solution, wherein the student did not implement such an attribute, and instead iterates through the Stack's internal array to count the number of non-zero elements. Note that the class implementation is ommitted for presentation in this dissertation, and only the *size* method is presented. We thus evaluate their solution by applying the tests defined in Figure 4.8b, which compare not only the result of invoking the student's *size* function but also considers the number of loop iterations (`CountLoopIterations`) and of array read operations (`CountArrayReadAccesses`). The test outputs, shown in Figure 4.8d, show

that the student's solution followed incorrect algorithm behaviour even though a correct result was produced.

(A) Reference solution (Average.java).

```
static double average(double[] a) {
    double sum = 0.0;
    for (int i = 0; i < a.length; i++)
        sum += a[i];
    return sum / a.length;
}
```

(B) DSL test suite.

```
val tests = TestSuite("path/to/reference/Average.java") {
    Case(CountLoopIterations()) {
        call("average", listOf(1,2,3,4,5), expected = 3.0)
        call("average", listOf(0,2,3,5,7), expected = 3.4)
    }
}
```

(C) Solution under testing (Solution.java) — with a defect, starting at index 1.

```
static double average(double[] a) {
    double sum = 0.0;
    for (int i = 1; i < a.length; i++)
        sum += a[i];
    return sum / a.length;
}
```

(D) Invoking the execution of a test suite to a solution under evaluation.

```
val results: List<ITestResult> = tests.apply(
    subjectPath = "path/to/Solution.java"
)
results.forEach { println("$it\n") }
```

(E) Output of the test results.

```
[fail] average([1, 2, 3, 4, 5])
  Expected: 3.0
  Found: 2.8

[fail] average([1, 2, 3, 4, 5])
  Expected loop iterations: 5
  Found: 4

[pass] average([0, 2, 3, 5, 7])
  Expected: 3.4

[fail] average([0, 2, 3, 5, 7])
  Expected loop iterations: 5
  Found: 4
```

FIGURE 4.7. Assignment evaluation scenario using Witter's API.

(A) Reference solution's *size* method (StackReference.java).

```
public int size() {
    return this.size; // Auxiliary integer attribute
}
```

(B) DSL Test Suite.

```
val tests = TestSuite("path/to/reference/StackReference.java") {
    Case {
        val stack = ref { new("Stack", 5) }
        call("push", stack, 1)
        call("push", stack, 2)
        call("push", stack, 3)
        using (CountLoopIterations() + CountArrayReadAccesses()) {
            call("size", stack, expected = 3)
        }
    }
}
```

(C) Solution under testing with a defect — counts non-zero items individually (StackSolution.java).

```
public int size() {
    int s = 0;
    for (int i = 0; i < stack.length; i++)
        if (stack[i] != 0)
            s += 1;
    return s;
}
```

(D) Test results.

```
[pass] size(Stack(stack=[1, 2, 3, 0, 0], size=3))
  Expected: 3

[fail] size(Stack(stack=[1, 2, 3, 0, 0], size=3))
  Expected loop iterations: 0
  Found: 5

[fail] size(Stack(stack=[1, 2, 3, 0, 0], size=3))
  Expected array reads: 0
  Found: 5
```

FIGURE 4.8. Stack (data structure) testing using Witter's DSL.

4.3. Implementation

As mentioned in Chapter 3, Witter utilises Strudel as an execution environment for both the reference solutions and the solutions under testing. The evaluation is achieved by implementing a listener for the Strudel virtual machine that collects and aggregates the necessary information from the relevant virtual machine events. As both the reference solution and the students' code are executed, this listener captures code execution events and stores the data for each metric for each executed test case. This allows Witter to maintain a data structure which maps each executed procedure within a test case to its corresponding runtime metric data (e.g. the number of loop iterations, etc.), enabling

```
grammar TSL; // Test Specification Language

specification: (annotation (annotation)*)? EOF;

annotation:
    '@Test' '(' args=(TEST_ARGUMENTS | INTEGER) ')'
    | '@CountLoopIterations' ('(' margin=INTEGER ')')?
    | '@CheckObjectAllocations'
    | '@CheckArrayAllocations'
    | '@CountArrayReads' ('(' margin=INTEGER ')')?
    | '@CountArrayWrites' ('(' margin=INTEGER ')')?
    | '@CheckSideEffects'
    | '@CountRecursiveCalls' ('(' margin=INTEGER ')')?
    ;

INTEGER: [0-9]+;

IDENTIFIER: [a-zA-Z_$][a-zA-Z0-9_$]*;

TEST_ARGUMENTS: TEST_ARGUMENT (TEST_ARGUMENT)*;
TEST_ARGUMENT: ~('(' | ')' | '\n' | '\r');

IGNORE: (WHITESPACE | COMMENT | NEWLINE) -> skip;
fragment NEWLINE: '\n' | '\r' | '\r\n';
fragment WHITESPACE: (' ' | '\t')+;
fragment COMMENT: '#*' .+? '*#';
```

FIGURE 4.9. ANTLR grammar for code annotation syntax.

the comparison between procedures for both solutions. Once the execution has finished, Witter utilises this data structure to construct the test result messages for each executed test case by comparing the measured values relating to the reference solution and the student's solution.

The language for annotating code solutions is implemented by a two-step parsing and translation process. A simple grammar was defined using ANTLR[1], a parser generator library, so that the annotations could be parsed and subsequently translated to Witter's own internal model for representing test specifications. This grammar is presented in Figure 4.9. The translation process parses and translates each annotation and its respective arguments, with the exception of the arguments of the `@Test` annotation, which are temporarily stored as simple strings. These are parsed by the JavaParser library, and in turn, translated to Strudel's model for passing the arguments to the execution process.

The DSL is implemented in Kotlin using features such as function literals with receivers[2] and type-safe builders[3]. These features enable the implementation of DSLs in a semi-declarative way, allowing the syntax of Witter's DSL to resemble that of a usual programming language, aligning with our goal of providing instructors with a low-effort way to specify test cases.

[1] https://www.antlr.org/
[2] https://kotlinlang.org/docs/lambdas.html
[3] https://kotlinlang.org/docs/type-safe-builders.html

Internally, Witter represents the information regarding test case specifications using a set of constructs which define the abstract syntax of what could be considered a simple programming language, which we refer to as Witter's *Test Specification Language*. Both the comments used to annotate code solutions and the DSL serve as front-end syntaxes for instantiating this language. Witter, in turn, acts as an interpreter which executes the testing directives specified and applies them to the solution under testing.

Currently, Witter's testing functionality supports instantiating objects and their manipulation through instance methods, as well as the invocation of static methods present in the code under evaluation. The results of invoking procedures and creating objects can be stored in variable references, which can be used to chain these calls in order to specify stateful testing scenarios.

The implementation of the test specification language is tailored towards a balance between the robustness and simplicity of test specification. The possibility of chaining procedure calls and nesting test cases unlocks the possibility of minimising the code necessary to define sets of test cases offering good coverage of what is standard to assess in static or object-oriented contexts of introductory programming. Furthermore, a structured internal model for representing test specifications enables the potential to create additional front-end syntaxes for instantiating these specifications, should such prove relevant in any future work regarding the library.

CHAPTER 5

Evaluation

In order to explore an answer to Research Question **RQ2**, we conducted an experiment using the proposed DSL to evaluate student assignments. Moreover, the application of a set of real student submissions allows us to evaluate Witter's coverage of supported Java language features in the context of introductory programming. We now describe the context and methodology of this evaluation, and present its results.

5.1. Context

We collected a set of 2,389 student assignment submissions spanning two offerings of the Algorithms and Data Structures course taken by first year undergraduate students of Computer Science and Engineering and related bachelor's degrees. The submission process was independent from our approach, hence no constraints were posed regarding Witter's limitations. For this reason, some submissions could not be handled. We analyzed five distinct assignments with the following guidelines:

(1) Implement three different classes, each solving the dynamic connectivity (union-find) problem using different approaches: Quick-Find, Quick-Union, and Weighted Quick-Union with Path Compression.
(2) Implement a Queue data structure supporting the String data type. Use a circular resizing array implementation to store the data internally without needing to limit the queue's memory capacity *a priori*.
(3) Implement an optimised version of the Insertion Sort algorithm which executes fewer array access operations by avoiding swap operations and instead shifting elements directly one position to the right as needed.
(4) Implement a generic List data structure utilising a dynamic sequence of simply-linked nodes internally to store elements.
(5) The implementation of the Heap Sort algorithm seen in the lectures assumes the array indices begin at 1. Modify the algorithm's implementation to support sorting arrays starting at index 0.

Each assignment was given to students in this order throughout the semester. The guidelines were translated from Portuguese, which included more API details that have been truncated for presentation in this paper.

Table 5.1 presents the number of valid submissions used for evaluating the DSL for each assignment. We consider a submission to be valid if the submitted file matches the name and extension indicated in an assignment's guidelines, and if Java can successfully compile the source code. The inconsistent number of submissions is explained by two

TABLE 5.1. Number of student submissions used for evaluating Witter's DSL.

Assignment	**Description**	**Total**	**Valid**
A1	Dynamic Connectivity	584	424 (72.6%)
A2	Resizing Array Queue of Strings	573	490 (85.5%)
A3	Improved Insertion Sort	212	120 (56.6%)
A4	Generic Linked List	550	476 (86.5%)
A5	Heap Sort	470	266 (56.6%)
		2389	**1776 (74.3%)**

factors: our observations of a growing rate of absenteeism as the semester progresses; and, every assignment being present in the two course offerings considered except assignment A3, which was only present in a single offering.

5.2. Method

Figure 5.1 presents the test specifications for each assignment using Witter's DSL. Each student submission was evaluated using Witter through the corresponding test specification, and the results of the evaluation were processed to find cases where the student's implementation passed black-box tests but failed white-box tests. Given the nature of the chosen assignments, the evaluation focused on the metrics for counting loop iterations, array read and write access operations, allocated memory (for resizing array operations), and argument side effects (for checking whether sorting algorithms effectively sorted the input array).

In order to assess Witter's suitability for large-scale assessment processes, we measure the average execution time for the evaluation of each submission. The execution took place in a laptop system with a 12-core 2.6GHz Intel i7-9750H CPU and 16GB of RAM. While a completely isolated simulation is impossible in a standard system, care was taken to minimise the impact of other operating system processes on the performance of Witter's execution.

It is usually the case that computer engineering students have a general propensity to cheat in programming assignments [**1**]. We took this factor into consideration during Witter's evaluation by running plagiarism analysis on all student submissions using JPlag[1], a plagiarism checking tool resistant to a broad range of obfuscation techniques and which provides easily-interpretable results of its analysis [**40**, **41**]. While a connection between plagiarism checking and white-box assessment has not been observed, we include this analysis as a means to safeguard our evaluation against possible accidental biases stemming from students having plagiarised the same incorrect sources (e.g. copying from a fellow classmate who made mistakes). The results of this analysis are given in Section 5.3, with Table 5.3 providing an overview of the code similarity between the students' submissions for each assignment.

[1] `https://github.com/jplag/JPlag`

(A) Test specification for assignment A1.

```
Case(CountLoopIterations(1) + CountArrayWriteAccesses(1) +
CountArrayReadAccesses(1)) {
    val uf = ref { new("QuickFindUF", 100) }
    val connected = mutableListOf<Pair<Int, Int>>()
    (1 .. 100).forEach { _ ->
        val p = (100 * random()).toInt()
        val q = (100 * random()).toInt()
        uf.call("union", p, q)
        connected.add(Pair(p, q))
    }
    connected.forEach { uf.call("connected", it.first, it.second) }
}
```

(B) Test specification for assignment A2 (excerpt).

```
Case(CountLoopIterations(1), "testDequeue") {
    val queue = ref { new("Queue") }
    queue.call("enqueue", "witter")
    queue.call("enqueue", "is")
    queue.call("enqueue", "cool")
    queue.call("dequeue")
    queue.call("dequeue")
    queue.call("dequeue")
}
Case(CountLoopIterations(1) + CountMemoryUsage()) {
    val queue = ref { new("Queue") }
    (1..100).forEach { queue.call("enqueue", it.toString()) }
}
```

(C) Test specification for assignment A4 (excerpt).

```
Case(CountLoopIterations(2) + CheckObjectAllocations, "testSize") {
    val list = ref { new("List") }
    list.call("size")
    list.call("add", "hello")
    list.call("size")
    list.call("add", "world")
    list.call("size")
}
```

(D) Test specification for assignments A3 and A5.

```
Case(CheckSideEffects + CountLoopIterations(1) +
CountArrayWriteAccesses(1) + CountArrayReadAccesses(1)) {
    call("sort", listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
    call("sort", listOf(7, 3, 2, 1, 5, 6, 10, 8, 9, 4))
    call("sort", listOf(7.32, 3.14, 2.14, 1.93, 5.99, 6.74, 10.21,
                        8.84, 9.26, 4.56))
    call("sort", listOf("sorting", "algorithms", "are", "really",
                        "very", "cool"))
    call("sort", listOf(10, 9, 8, 7, 6, 5, 4, 3, 2, 1))
}
```

FIGURE 5.1. Test specifications for the evaluated assignments (A1–5).

TABLE 5.2. Number of black-box and white-box passes and failures for loaded submissions. Loaded submissions are given as a % of the Valid submissions, and Passes/Failures are given as a % of the Loaded submissions. Execution Time is the average over all submissions of the corresponding assignment.

Assignment	**Valid**	**Loaded**	**Blackbox Pass**	**Whitebox Fail**	**Time**
A1	424	413 (97.4%)	407 (98.5%)	79 (19.1%)	850 ms
A2	490	433 (88.4%)	391 (90.3%)	98 (22.6%)	300 ms
A3	120	110 (91.7%)	107 (97.3%)	49 (44.5%)	20 ms
A4	476	347 (72.9%)	347 (100%)	32 (9.22%)	100 ms
A5	266	233 (83.8%)	221 (99.1%)	58 (26.0%)	30 ms
	1776	**1526 (85.9%)**	**1473 (96.5%)**	**316 (20.7%)**	

5.3. Results

Table 5.2 summarizes the evaluation results. Witter successfully loaded a total of 1,526 student submissions from the 1,776 valid submissions described in Table 5.1, constituting approximately 86% of all valid submissions. For these, all the assignments had a black-box tests pass rate greater than 90%. However, the white-box tests failure rate ranged approximately between 9% and 45%, with assignment A4 exhibiting the largest failure rate.

Witter failed to load 250 student submissions, which constitutes approximately 14.1% of the valid submissions. This limitation stems for the Java translation module used by the Strudel library, which does not implement translation features for the complete Java language specification. Additionally, the JavaParser library does not support some features of more recent Java versions which were used by some students, such as `switch` statements.

Figure 5.2 presents the distribution of white-box metrics that produced failures in each assignment. Approximately 50% of failed assertions for assignment A1 were caused by an incorrect number of array read operations, with the majority of the remaining errors relating to an incorrect number of array write operations. Out of the considered metrics, assignments A2, A3, and A5 contained white-box errors relating to the number of loop iterations and array read and write operations, the proportion of which are approximately equal, each constituting approximately one third of each assignment's detected errors. Assignment A4 produced the simplest results, with all failed assertions relating to the number of loop iterations. While a comprehensive analysis of all student submissions with white-box test failures was not conducted, a brief view of those submissions for assignment A3 seemed to indicate that students might somewhat struggle with understanding the overall goal of the assignment, with many having implemented the base Insertion Sort algorithm with minor syntactic differences as opposed to implementing the desired algorithmic behaviour changes. Some submissions also indicate some difficulty in thinking of array traversals with 0-based indices, especially when multiple loop variables are interdependent, in particular

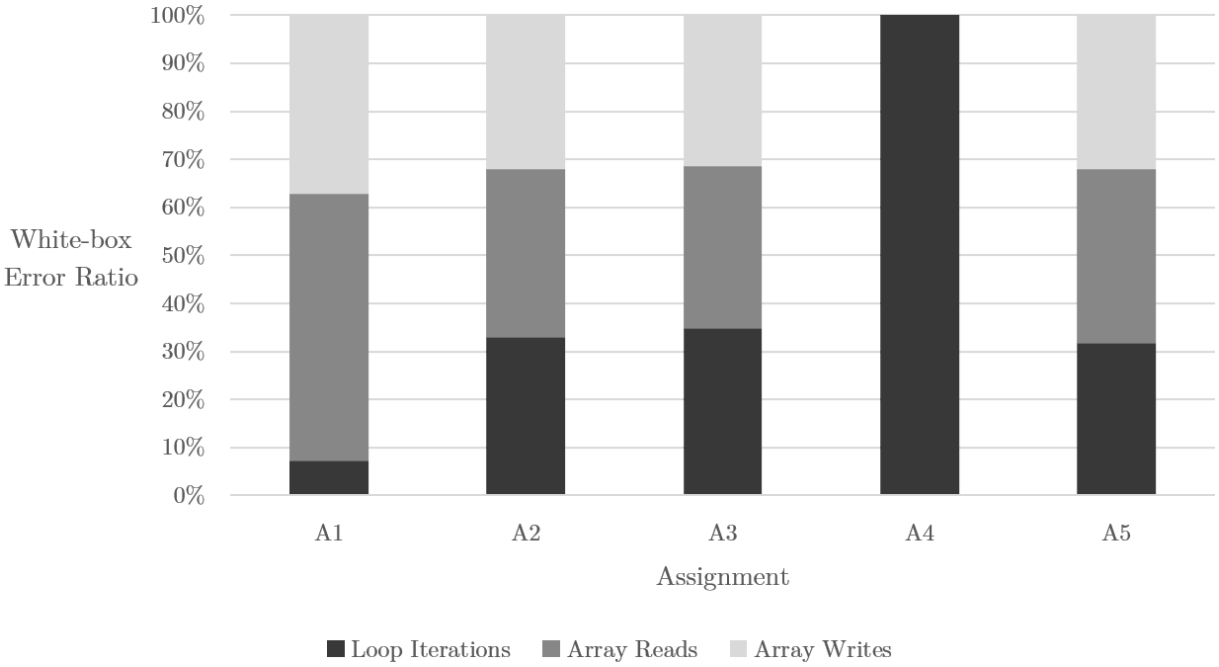

FIGURE 5.2. Ratio of each white-box failure type per assignment.

indicating that some students have trouble discerning where to start and end their loops in a way that avoids out-of-bounds exceptions occurring when nested iterations are required.

Of the considered white-box metrics, those whose measured values are integers, namely loop iterations, array read operations, and array write operations, have a numerical margin of error relative to the reference solution within which submissions must lie to be considered correct, as described in Section 4.1. This enables us to analyse the amount by which, for each metric, each failed assertion on students' submissions deviated from the respective reference solution. We thus present in Figure 5.3 the distributions of these deviations. Note that some data had to be filtered from this visualisation, as some submissions presented apparent errors relating to the white-box metrics but which were in truth caused by the students' code exceeding the Strudel virtual machine's loop iteration limit, effectively being timed out. Nevertheless, the valid data is significant enough to observe that, for assignments A2 and A4, the devations from the reference solutions are considerably low, while for the remaining assignments, some of the evaluated metrics deviated by a considerable margin from the reference solution, indicating implementations which are significantly different from the intended solutions.

Our suspicions regarding code similarity were confirmed by JPlag's analysis, with the average similarity between submissions ranging approximately from 20% to 64%, as summarised in Table 5.3. While an average code similarity between submissions of approximately 20%, as for assignment A3, can fall within reasonable expectations for an assignment of this scope, values of 44% or 61% average similarity, as seen for assignments A5 and A1, respectively, begin to raise concerns of considerable plagiarism and how it can affect the results of our analysis. Care should be taken in future work to guarantee

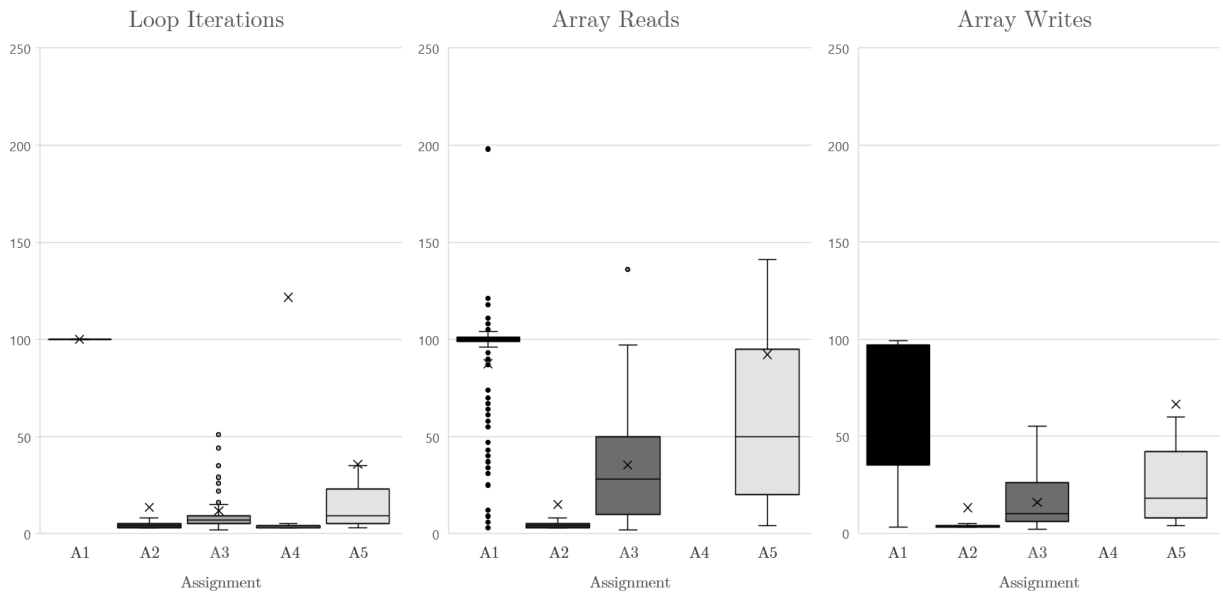

FIGURE 5.3. Deviations from reference solution for each failed assertion for each white-box metric. Submissions which failed because they exceeded the virtual machine's iteration limit (timed out) are not illustrated.

that code similarity does not skew the results by introducing an accidental bias relating to students unknowingly plagiarising from incorrect sources.

TABLE 5.3. Average submission code similarity per assignment.

Assignment	**Average ± Std. Dev.**
A1	64.2 ± 20.0%
A2	25.7 ± 17.5%
A3	19.8 ± 17.6%
A4	22.6 ± 14.0%
A5	44.1 ± 24.0%

The results of the evaluation show that a considerable number of students produce faulty implementations when it comes to algorithmic behaviour, which would go unnoticed by an evaluation process focusing only on the results produced by the students' code. Even the assignment which contained the least faulty implementations, assignment A4, shows that 32 students, which is a significant amount, could be misled by a black-box-only assessment process. Furthermore, in assignment A3, which required arguably the most originality (as further indicated by the lowest average code similarity among the considered assignments) in developing an alternative version of a standard algorithm, we saw a 44.5% failure rate, which constitutes nearly half of all considered submissions. We can thus conclude that the usage of an assessment tool which provides information about the execution of students' implementations is useful, not only from the perspective of an instructor aiming to accurately grade student submissions, but also to prevent misleading students by informing them of the correctness of their implementation's results regardless of how those results were produced.

The evaluation further allows us to observe that our current work on Witter and Strudel displays an acceptable coverage of standard Java constructs and language features, as seen by the overall 85.9% successful file loading rate seen during the evaluation phase, and the 72% or higher file loading rate within each assignment. Nevertheless, further work should be carried out to extend the scope of supported functionalities and therefore enable the usage of more diverse assignments for a more detailed evaluation.

The average time taken to execute the evaluation of each submission demonstrates Witter's acceptable performance for usage in educational contexts, with each submission taking an average of under one second to be evaluated in the scope of the considered assignments. We note that the execution time is dependent on the complexity of the specified tests, with longer or more comprehensive tests corresponding to a longer execution time. For the considered assignments, we note that the noticeable discrepancy in the average execution time per submission is due to the fact that some assignments, namely A1, A2, and A4, perform a higher number of method calls than assignments A3 and A5. In particular, the test case for assignment A1 executes 200 method calls in total, as shown in Figure 5.1a. Similarly, one of the test cases for assignment A2, as shown in Figure 5.1b, executes 100 method calls. The test case for assignment A4 executes a total of 28 procedure calls, 5 of which are shown in Figure 5.1c. Finally, assignments A3 and A5 were evaluated with the simplest test case, with each executing only 5 procedure calls. We thus observe a clear relation between the number of executed procedure calls per test case and the average time taken to evaluate each submission.

5.4. Threats to Validity

The main limitations of the evaluation come from the scope (number of student submissions), the context (all students from the same institution), and the selected dataset (limited scope of assignments). Care should thus be taken in any subsequent evaluations to address these limitations. For instance, taking Strudel's supported functionalities into account during the submission process, for example by developing assignment guidelines such that the required Java language features fall within Strudel's currently-supported subset of Java, or by placing stronger focus on assignments' specific algorithm behaviour requirements, could provide a larger usable dataset of submissions, enabling a more significant evaluation.

When collecting student assignments, we observed that a significant portion of the required solutions was provided almost directly in the available course materials, which could hinder student originality by motivating them to simply replicate or complete (partial or full) solutions which are already available, which could skew the results of our evaluation. This might already be visible in the considerable average similarity of the submissions for some assignments. Care should be taken to mitigate this in our future work, possibly by conducting a more controlled submission process (e.g., assignments solved by students in class under instructor supervision).

CHAPTER 6

Conclusions

The development of the work described in this dissertation was motivated by the identification of an important problem in the field of automated programming assessment. Namely, existing automated assessment tools suffer from limited functionality when it comes to the evaluation of students' submissions from dynamically-collected execution data. We hypothesised that an assessment tool enabling the collection of white-box execution data from students' submitted code solutions could serve to benefit existing automated assessment systems, and consequently proposed Research Questions **RQ1** and **RQ2** to assess the validity of this hypothesis.

We arrive at a positive answer to Research Question **RQ1** through our successful implementation of Witter, as seen in Section 4.1. Our implementation demonstrates the feasibility not only of implementing a software library that aligns with our proposed goals of providing an easy-to-use augmentation tool for current assessment systems, but also one which does so in an effective manner and is thus adequate for usage in educational environments. Being provided as a software library implemented in Kotlin, Witter is straightforward to integrate into JVM-based automated assessment systems. The scalability of our implementation is shown by its usage of Strudel, which provides an infrastructure capable of being adapted to other programming languages, provided that a translation module for that language can be developed.

When it comes to Research Question **RQ2**, the application of Witter to real student assignments offers practical insights that successfully motivate the usefulness of a tool like Witter in introductory programming contexts. Namely, we observe that a significant number of standard student submissions could be subject to erroneous evaluations if the evaluation fails to consider the internal behaviour of the students' code during its execution. In particular, students seemed to display a more significant number of algorithm behaviour errors for assignments which required greater originality or which required the most change from their standard implementation, even if the end result of the algorithm's application was identical to the corresponding reference solution. The results of our evaluation thus provide practical evidence that basic evaluation methods might not be sufficient for adequately evaluating students at an introductory programming level, highlighting the possible benefits of dynamic white-box assessment for automated programming assessment and its scarcity in existing assessment systems.

6.1. Limitations and Future Work

We continue to envision Witter as a tool not only aiming to be integrated into existing automated assessment systems as a way to extend the scope of their assessment functionalities, but also as courseware to be used by introductory programming students in a classroom environment, providing a learning process augmented through instant feedback on their attempts to solve programming exercises. We have not yet been able to conduct a user study with programming students or instructors, and this is undoubtedly the main focus for potential future work as the scope and stability of both Witter and Strudel's functionalities increase. Namely, a study with introductory programming students is necessary to gauge whether Witter is useful for its envisioned context. Furthermore, a study with programming instructors could offer insight into the effort required to develop assignments adopting Witter's DSL, allowing us to assess whether the proposed methods effectively offer a practical advantage over existing solutions and to address limitations identified by instructors using Witter towards its idealised goal of being used as an automated assessment tool. Finally, given a longer time frame, a study could be conducted in the context of an introductory programming course to analyse the long-term effect on the usage of Witter or similar assessment tools in students' learning outcomes. One possibility would be to introduce students to Witter in an introductory programming course and assess how its usage affects their learning outcomes not only for that course, but for subsequent programming courses.

We envision a potential use case for Witter as a tool to augment systems which generate questions about students' understanding of their own code. Since systems which generate personalised questions for a student's code can aid the student in consolidating their knowledge [**31**, **39**], we envision that such a tool could be augmented with the information provided by Witter to generate more comprehensive questions or provide more detailed feedback on incorrect answers. A code generation system augmented with Witter could provide a fine-grained, personalised teaching tool to aid introductory programming students in consolidating concepts and strengthening their code comprehension skills. A similar use case could consist in using the detailed program execution information to aid in generating hints about student code [**14**, **17**].

Relating to Witter's functionalities, further work could be conducted in implementing more observable code execution events, enabling the computation of new metrics, such as the number of expressions evaluated, exchanges between array elements, or element comparisons, which are relevant in contexts such as algorithmic complexity analysis. Similarly, the addition of features to extend Strudel's functionality could broaden the scope of programs that Witter can assess. The Java translator used by the Strudel library could be developed further to support a larger subset of the Java language, which would also enable the use of more varied assignments.

A more unbiased evaluation of Witter's efficacy and relevance could be carried out by utilising an external, publicly-available dataset of programming student submissions,

guaranteeing the usage of code produced by students outside of our institution, which could also tackle possible issues of cheating or plagiarism. Moreover, a more refined evaluation which takes Strudel and Witter's functionalities into account could further address the quality of the assignments' specifications. From an instructor's standpoint, an analysis of the test results provided by Witter could aid in refining assignment guidelines to help direct students towards those aspects in which they show weaker understanding.

References

[1] C.L. Aasheim, Paige Rutner, L. Li, and S.R. Williams. Plagiarism and programming: A survey of student attitudes. *Journal of Information Systems Education*, 23:297–314, 01 2012.

[2] Tapio Auvinen. Harmful study habits in online learning environments with automatic assessment. In *Proceedings of the 2015 International Conference on Learning and Teaching in Computing and Engineering*, LATICE '15, page 5057, USA, 2015. IEEE Computer Society.

[3] Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. The effect of a web-based coding tool with automatic feedback on students' performance and perceptions. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 27, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Jan vom Brocke, Alan Hevner, and Alexander Maedche. *Introduction to Design Science Research*, pages 1–13. 09 2020.

[5] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[6] Afonso Caniço and André Santos. Witter: A library for white-box testing of introductory programming algorithms. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E 2023, pages 69–74, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Afonso Caniço and André Santos. A domain-specific language for dynamic white-box evaluation of java assignments. In *Proceedings of the 5th International Computer Programming Education Conference (to appear)*, ICPEC 2024, 2024.

[8] Angela Carbone, Jason Ceddia, Simon, Daryl D'Souza, and Raina Mason. Student concerns in introductory programming courses. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ACE '13, page 4150, AUS, 2013. Australian Computer Society, Inc.

[9] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[10] Shigeru Chiba. Load-time structural reflection in java. In Elisa Bertino, editor, *ECOOP 2000 — Object-Oriented Programming*, pages 313–336, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[11] Bruno Pereira Cipriano and Pedro Alves. Llms still can't avoid instanceof: An investigation into gpt-3.5, gpt-4 and bard's capacity to handle object-oriented programming assignments. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 162169, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Sébastien Combéfis. Automated code assessment for education: Review, classification and perspectives on techniques and tools. *Software*, 1(1):3–30, 2022.

[13] Pedro Crispim, Antónia Lopes, and Vasco T. Vasconcelos. Runtime verification for generic classes with congu2. In *Proceedings of the 13th Brazilian Conference on Formal Methods: Foundations and Applications*, SBMF'10, page 3348, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Alcino Cunha, Nuno Macedo, José Creissac Campos, Iara Margolis, and Emanuel Sousa. Assessing the impact of hints in learning formal specification. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 151161, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Robertas Damaeviius and Vytautas Stuikys. Taxonomy of the fundamental concepts of metaprogramming. *Information technology and control*, 37, 01 2013.

[16] Mohd Ehmer and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3, 06 2012.

[17] Jorge A. Gonçalves and André L. Santos. Jinter: A hint generation system for java exercises. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 375381, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Sebastian Gross and Niels Pinkwart. Towards an integrative learning environment for java programming. In *2015 IEEE 15th International Conference on Advanced Learning Technologies*, pages 24–28, July 2015.

[19] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. Pedal: An infrastructure for automated feedback systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, pages 1061–1067, New York, NY, USA, 2020. Association for Computing Machinery.

[20] John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

[21] Michael T. Helmick. Interface-based programming assignments and automatic grading of java programs. *SIGCSE Bull.*, 39(3):63–67, jun 2007.

[22] J. Holland, Antonija Mitrovic, and Brent Martin. J-latte: a constraint-based tutor for java. 01 2009.

[23] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, dec 2004.

[24] David Insa and Josep Silva. Automatic assessment of java code. *Computer Languages, Systems & Structures*, 53:59–72, 2018.

[25] Julian Jansen, Ana Oprescu, and Magiel Bruntink. The impact of automated code quality feedback in programming education. 2018.

[26] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.*, 19(1), sep 2018.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 327–354, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[28] Päivi Kinnunen and Beth Simon. My program is ok – am i? computing freshmen's experiences of doing programming assignments. *Computer Science Education*, 22(1):1–28, 2012.

[29] José Paulo Leal and Fernando Silva. Mooshak: a web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

[30] Teemu Lehtinen, Charles Koutcheme, and Arto Hellas. Let's ask ai about their programs: Exploring chatgpt's answers to program comprehension questions. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 221232, New York, NY, USA, 2024. Association for Computing Machinery.

[31] Teemu Lehtinen, Andre L. Santos, and Juha Sorva. Lets ask students about their programs, automatically. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, May 2021.

[32] Marko V. Lubarda, Alex M. Phan, Aidan Daniel Carrigg, Karthik Srinivasan, and Josephine Relaford-Doyle. Effect of automated instantaneous feedback, unlimited submission attempts, and optional exercises on student engagement, performance, and academic integrity in an introductory computer programming course for engineers. In *2023 ASEE Annual Conference & Exposition*, Baltimore, Maryland, June 2023. ASEE Conferences. https://peer.asee.org/43228.

[33] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[34] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. Automated grading and feedback tools for programming education: A systematic review. *ACM Trans. Comput. Educ.*, dec 2023. Just Accepted.

[35] Samim Mirhosseini, Austin Z. Henley, and Chris Parnin. What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 291297, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Joydeep Mitra. Studying the impact of auto-graders giving immediate feedback in programming assignments. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 388394, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Luís Nunes et al. Use of programming aids in undergraduate courses. In *Proceedings of the 5th International Computer Programming Education Conference (to appear)*, ICPEC 2024, 2024.

[38] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Trans. Comput. Educ.*, 22(3), jun 2022.

[39] André Santos, Tiago Soares, Nuno Garrido, and Teemu Lehtinen. Jask: Generation of questions about learners' code in java. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, ITiCSE '22, page 117123, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Timur Sağlam, Moritz Brödel, Larissa Schmid, and Sebastian Hahner. Detecting automatic software plagiarism via token sequence normalization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[41] Timur Salam, Sebastian Hahner, Larissa Schmid, and Erik Burger. Obfuscation-resilient software plagiarism detection with jplag. In *46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion. Institute of Electrical and Electronics Engineers (IEEE), 2024.

[42] Edward Sykes. Design, development and evaluation of the java intelligent tutoring system. *Technology, Instruction, Cognition and Learning*, 8:25–65, 01 2010.

[43] Manuel Sánchez, Päivi Kinnunen, Cristóbal Flores, and J. Ángel Velázquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31:453460, 02 2014.

[44] Eddy van den Aker and Ebrahim Rahimi. Design principles for generating and presenting automated formative feedback on code quality using software metrics. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 139150, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Fabienne M. van der Kleij, Theo J.H.M. Eggen, Caroline F. Timmers, and Bernard P. Veldkamp. Effects of feedback in a computer-based assessment for learning. *Computers & Education*, 58(1):263–272, 2012.

[46] Anne Venables and Liz Haywood. Programming students need instant feedback! In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 267–272, AUS, 2003. Australian Computer Society, Inc.

[47] Yuankai Xue, Hanlin Chen, Gina R. Bai, Robert Tairas, and Yu Huang. Does chatgpt help with introductory programming?an experiment of students using chatgpt in cs1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 331341, New York, NY, USA, 2024. Association for Computing Machinery.

[48] Ramón Zatarain Cabada, María Lucía Barrón Estrada, Francisco González Hernández, and Raúl Oramas Bustillos. An affective learning environment for java. In *2015 IEEE 15th International Conference on Advanced Learning Technologies*, pages 350–354, 2015.