Witter: A Library for White-Box Testing of Introductory Programming Algorithms

Afonso B. Caniço
ambco@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL)
Portugal

André L. Santos
andre.santos@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR-IUL
Portugal

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. We present Witter, a software testing library that allows programming educators to define white-box tests for Java source code. Our tests analyze 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 behavior.

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Software and its engineering** → **Software testing and debugging**.

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

ACM Reference Format:
Afonso B. Caniço and André L. Santos. 2023. 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 '23), October 25, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3622780.3623650

1 Introduction

Software testing is an integral part of software development, allowing for the assessment of whether or not a given application verifies the requirements or produces acceptable results. By far, the most common method of performing software tests relies on black-box testing, where only the outputs 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, they do not provide any means to analyze or evaluate non-trivial aspects of program execution, such as efficiency, memory usage, or algorithm behavior (that is, if an algorithm executes internally as expected).

When learning a programming language, it is generally regarded that students benefit from informative and constructive feedback that allows them to explore their mistakes and deepen their understanding [16]. Moreover, it is widely agreed upon that several aspects need to be taken into consideration when grading students' programming assignments [2], which most assessment systems do not support [14]. Since students respond more positively when they manage to stay on track and autonomously arrive at the solution to a problem [13], it is also imperative that the provided feedback offers sufficient guidance without compromising the students' feelings of autonomy.

While extensive research and development work has been conducted on automated programming analysis and evaluation systems for educational environments, most of the available tools are either based on black-box unit testing (e.g. JUnit), static code analysis (e.g., Semmle[2]), or a combination of both techniques (e.g., [17]), with approaches focusing on the dynamic collection of white-box metrics relating to code execution being markedly uncommon [14].

In this paper, we present Witter[3], a novel software testing library with educational purposes that provides an infrastructure for defining and running white-box tests, supporting a subset of Java's syntax. Tests are described by annotating code solutions of exercises with simple directives that define what should be tested. Writing Witter tests does not require any sort of program instrumentation skills, as the collection in-depth metrics about the behavior of programs is expressed with high-level directives. In addition to a pass/fail flag, the output of tests includes messages concerning mismatches regarding what was expected to happen during execution.

SPLASH-E '23, October 25, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0390-4/23/10.
https://doi.org/10.1145/3622780.3623650

[1] https://junit.org
[2] https://semmle.com/
[3] https://github.com/ambco-iscte/witter

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 exercise goals are met. When such tests fail, users will be provided with constructive formative feedback to improve submissions. On the other hand, white-box tests could be integrated into exercise development environments (in addition to the black-box tests). The test results and feedback messages raise awareness of exercise goals that are not fulfilled, which otherwise could go unnoticed if the return values are correct. Both of these use cases could help students on achieving autonomous learning paths toward the desired practice goals.

2 Related Work

Being 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 (e.g., AspectJ[12][4], Javassist [3][5], or ASM[6]).

FindBugs[9][7] 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.

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 [11, 14]. Efforts have been made to develop software testing solutions that not only simplify the process on the instructor's side but 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 [4, 11].

JavAssess [10], similarly to Witter, is a Java library used to integrate deeper code analysis into existing automated assessment tools. Their approach, however, combines traditional unit testing with code instrumentation and meta-programming functionalities and does not offer a way to dynamically collect white-box execution metrics. AutoGrader [7] is a similar assessment library, leveraging on metaprogramming functionalities along with typical unit testing for code assessment.

Pedal [6] is an infrastructure supporting Python 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 this approach allows for feedback much more detailed than simple unit testing systems, it still disregards details of the program's execution.

Several intelligent tutoring systems (ITS), such as Java Sensei [18], FIT [5], JITS [15], or J-LATTE [8], have been proposed or developed for Java, supporting feedback generation systems whose goals align with those of our work. Among other components, an ITS requires automated assessment and feedback, a task that could be augmented with the testing capabilities of Witter.

3 Witter library

Witter aims at providing testing support for well-defined introductory programming exercises in Java, given as *reference solutions* by programming instructors. These are annotated using our *test specification language*. In turn, students submit solutions that are compared against the reference solutions using *runtime metrics*.

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 *Test Specification Language* (TSL), whose syntax 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.

Table 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 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 reference solution (most often ±1). If not specified, the *threshold* value is zero.

Figure 1a illustrates an example of specifying a Witter test for a simple function that calculates the sum of an array of integers. TSL's annotation `@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.

[4] www.eclipse.org/aspectj/
[5] www.javassist.org
[6] http://asm.objectweb.org/
[7] https://findbugs.sourceforge.net/

Table 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.

(a) Test specification example.

```
/*
@Test({1, 2, 3, 4, 5})
@Test({2, 4, 6})
@CountLoopIterations
@CountArrayReads
@CheckSideEffects
*/
public static int sum(int[] a) { ... }
```

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

```
Test test = new Test("ReferenceSolution.java")

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

Figure 1. Witter API for test specification and execution.

As Witter is designed for third-party integration, we provide a form of executing the tests programmatically (see Figure 1b). Tests are executed providing an annotated reference solution as described in Figure 1a, and a solution that one wishes to assess. The test results 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 1);
- the location of code elements involved in the failed tests (e.g., procedure, parameters, loop structures);
- a human-readable descriptive feedback message.

4 Examples

This section presents three examples of how Witter's white-box testing functionalities 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.

4.1 Factorial (recursive)

Consider the classical example of a recursive implementation of the factorial function as an exercise (see Figure 2a). In this case, the test specification 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 2b), 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 2c 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.

4.2 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 3a). The test specification defines two test cases, one positive and one

(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 2. Example: factorial testing using Witter.

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 3b), 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.

Despite that the test results indicate that both cases have the expected result (see Figure 3c), however, 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.

4.3 Insertion sort (procedure)

Consider an exercise to implement insertion sort as a procedure that modifies an array of integers (see Figure 4a). 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, in this case, we aim at checking that the behavior of the solution under testing

(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 3. Example: binary search testing using Witter.

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 4b), which performs a correct sorting, but through selection sorting. As with the previous example, here too, 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 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. Example: insertion sort testing with Witter.

5 Implementation

The core functionality of Witter relies on the Strudel library[8], which enables the execution of program models while performing fine-grained observation of execution events at the statement level. Witter captures these events in order to synthesize the relevant code metrics. Namely, it captures code execution events for procedure invocations and termination, loop iterations, object and array allocations, and array and variable assignments.

Strudel provides a virtual machine capable of executing models of programs, which in our case, are obtained by translation of Java code. We support a subset of Java's syntax, to the extent that we consider necessary in the context of introductory programming courses. Strudel itself executes code in a sandboxed manner, constraining the available instructions and safeguarding against scenarios such as infinite loops or out of memory errors. As such, the execution of Witter tests does not require any particular concerns regarding security.

The TSL is implemented by a two-step parsing and translation process. A simple grammar was defined using ANTLR[9] so that the annotations could be parsed and subsequently translated to Witter's own internal model for representing test specifications. This process parses and translates each annotation and their 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[10], and in turn, translated to Strudel's model for passing the arguments to the execution process.

6 Discussion

We envision two main use cases for Witter: a tool to enrich the capabilities of automated assessment systems; and, as courseware in introductory programming, integrated into a development environment where students are provided with more detailed, formative feedback to support the process of solving exercises. While we have not yet been able to conduct tests with students to measure the efficiency of such a tool in an educational environment, so far, our work demonstrates the practical feasibility of a foundational component to materialize these ideas.

Currently, the code that Witter can evaluate is limited to a subset of Java's features, a restriction imposed by the current implementation of Strudel. A large-scale study of BlueJ's Blackbox repository [1] of novice programmers' code revealed that the vast majority of users make use of a relatively small subset of the Java language. This supports our belief that introductory programming teaching only requires an elementary set of constructs to express algorithms, despite which programming language is being used. Therefore, we argue that working with a language subset is not a significant issue, as far as students are provided with adequate compiler messages when their code uses elements that are not supported.

Nonetheless, further work could be carried out to extend Strudel, and in turn, broaden the scope of programs that Witter can assess. Additionally, 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 and array swaps, which are relevant in contexts such as algorithmic complexity analysis.

[8] https://github.com/andre-santos-pt/strudel

[9] https://www.antlr.org/

[10] https://javaparser.org/

Many typical introductory programming assignments require the implementation of object-oriented solutions such as data structures. So far, Witter does not support object testing functionalities for this sort of exercise, including tests that require a sequence of several assertions over the same object or interaction between different objects. This is likely to be the next increment to Witter we will focus on.

We believe that students should have some freedom in adapting their implementations as long as they fall within the scope of the intended solutions, as forcing all students to conform to a specific, rigid solution contradicts the intent of our work of promoting student autonomy (not constraining it). We thus plan to implement a seamless way of specifying multiple, alternative reference solutions, as opposed to the testing process relying on a single reference solution.

Future work should be carried out to evaluate the impact of using Witter in a classroom environment, both as courseware and through automated assessment systems. Driven by usability shortcomings, we can further extend or refine current features to better adapt to students' learning patterns. As a first step, we plan to conduct a controlled experiment to investigate how well can students understand and make use of Witter's feedback.

Acknowledgments

This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

References

[1] Neil C. C. Brown, Pierre Weill-Tessier, Maksymilian Sekula, Alexandra Lucia Costache, and Michael Kölling. 2023. Novice Use of the Java Programming Language. *ACM Trans. Comput. Educ. (TOCE)* 23, 1 (2023), 10:1–10:24. https://doi.org/10.1145/3551393

[2] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. 2003. On automated grading of programming assignments in an academic institution. *Computers & Education* 41, 2 (2003), 121–131. https://doi.org/10.1016/S0360-1315(03)00030-7

[3] Shigeru Chiba. 2000. Load-Time Structural Reflection in Java. In *ECOOP 2000 — Object-Oriented Programming*, Elisa Bertino (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–336.

[4] Sébastien Combéfis. 2022. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* 1, 1 (2022), 3–30. https://doi.org/10.3390/software1010002

[5] Sebastian Gross and Niels Pinkwart. 2015. Towards an Integrative Learning Environment for Java Programming. In *2015 IEEE 15th International Conference on Advanced Learning Technologies*. 24–28. https://doi.org/10.1109/ICALT.2015.75

[6] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. 2020. Pedal: An Infrastructure for Automated Feedback Systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1061–1067. https://doi.org/10.1145/3328778.3366913

[7] Michael T. Helmick. 2007. Interface-Based Programming Assignments and Automatic Grading of Java Programs. *SIGCSE Bull.* 39, 3 (jun 2007), 63–67. https://doi.org/10.1145/1269900.1268805

[8] J. Holland, Antonija Mitrovic, and Brent Martin. 2009. J-Latte: a Constraint-Based Tutor for Java. (01 2009).

[9] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (dec 2004), 92–106. https://doi.org/10.1145/1052883.1052895

[10] David Insa and Josep Silva. 2018. Automatic assessment of Java code. *Computer Languages, Systems & Structures* 53 (2018), 59–72. https://doi.org/10.1016/j.cl.2018.01.004

[11] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (sep 2018), 43 pages. https://doi.org/10.1145/3231711

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 327–354.

[13] Päivi Kinnunen and Beth Simon. 2012. My program is ok – am I? Computing freshmen's experiences of doing programming assignments. *Computer Science Education* 22, 1 (2012), 1–28. https://doi.org/10.1080/08993408.2012.655091

[14] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (jun 2022), 40 pages. https://doi.org/10.1145/3513140

[15] Edward Sykes. 2010. Design, Development and Evaluation of the Java Intelligent Tutoring System. *Technology, Instruction, Cognition and Learning* 8 (01 2010), 25–65.

[16] Anne Venables and Liz Haywood. 2003. Programming Students NEED Instant Feedback!. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20* (Adelaide, Australia) *(ACE '03)*. Australian Computer Society, Inc., AUS, 267–272.

[17] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology* 55, 6 (2013), 1004–1016. https://doi.org/10.1016/j.infsof.2012.12.005

[18] Ramón Zatarain Cabada, Maria Barron Estrada, Francisco González Hernández, and Raul Oramas. 2015. An Affective Learning Environment for Java. (07 2015).

Received 2023-07-27; accepted 2023-08-24