


# Seven Years Later: Lessons Learned in Automated Assessment

Bruno Pereira Cipriano<sup>1</sup> ✉ 

COPELABS, Lusófona University, Lisbon, Portugal

Pedro Alves ✉ 

COPELABS, Lusófona University, Lisbon, Portugal

---

## Abstract

Automatic assessment tools (AATs) are software systems used in teaching environments to automatically evaluate code written by students. We have been using such a system since 2017, in multiple courses and across multiple evaluation types. This paper presents a set of lessons learned from our experience of using said system. These recommendations should help other teachers and instructors who wish to use or already use AATs in creating assessments which give students useful feedback in terms of improving their work and reduce the likelihood of unfair evaluations.

**2012 ACM Subject Classification** Applied computing → Computer-assisted instruction

**Keywords and phrases** learning to program, automatic assessment tools, unit testing, feedback, large language models

**Digital Object Identifier** 10.4230/OASICS.ICPEC.2024.3

**Funding** This research was funded by the Fundação para a Ciência e a Tecnologia under Grant No.: UIDB/04111/2020 (COPELABS).

## 1 Introduction

Automated assessment tools (AATs) are software tools used to automatically grade students' software solutions. These tools allow students to check if their code respects the assignment's requirements in terms of functional and/or code quality requirements, leading to increased student autonomy and reduced teacher workload [9, 13].

We have been using such a system since 2017 for multiple types of assessment, from small formative exercises to larger scale projects which have a large weight on the students' grades. Furthermore, we have employed automatic assessment in multiple courses, ranging from Introduction to Programming, Data-Structures and Algorithms, and Object-Oriented Programming (OOP).

Our team employs a common practice in terms of automatic assessment. The starting point of the process is a text-based description of the exercise(s) or project to solve. This description is sometimes complemented with auxiliary diagrams containing mathematical formulas, UML diagrams, and so on. Besides presenting the description, the assignment also informs students about any mandatory files, classes and/or functions, thus defining an Application Programming Interface (API). Finally, when students submit a solution<sup>2</sup>, their code is validated by a set of teacher-defined unit tests. For example, here's the assignment text for a typical exercise used in introductory programming courses: *Implement the function `static long sum(int n1, int n2)` which should return the sum of its two arguments.* Listing 1 shows an unit test which could be used to validate this assignment.

---

<sup>1</sup> corresponding author

<sup>2</sup> Assignments allow multiple submissions, promoting that students improve their work and learning outcomes.



## 3:2 Seven Years Later: Lessons Learned in Automated Assessment

■ **Listing 1** An example *JUnit* test for an exercise where students must implement a function which sums two integers. The feedback messages are only displayed when the respective assertion fails.

```
1 @Test
2 public void testSum() {
3     String feedback = "returned the wrong value";
4     assertEquals(5, Main.sum(1, 4), "sum(1, 4) " + feedback);
5     assertEquals(10, Main.sum(8, 2), "sum(8, 2) " + feedback);
6 }
```

A more complex OOP exercise could be the following: (...) *The GameManager class must have a static List<Player> getPlayers() function which returns a list with the players that are participating in the current game.* Listing 2 presents an example of a unit test for validating that requirement.

■ **Listing 2** An example *JUnit* test for an OOP project where students must implement a board game. The messages are only displayed when the respective assertion fails.

```
1 @Test
2 public void testGetPlayers01() {
3     GameManager manager = new GameManager("game-board-with-2-players.txt");
4     List<Player> players = manager.getPlayers();
5     assertNotNull(players, "getPlayers() should not return null");
6     String feedback = "getPlayers() returned a list of the wrong size";
7     assertEquals(2, players.size(), feedback);
8 }
```

We opted for this unit-tests-based approach because it is a more robust way of testing than the alternative of using “output matching”. Also, it is closer to current industry practices, with the added benefit of familiarising students with the types of tools that they will more likely use in their future careers.

Our experience over these last 7 years showed us that certain testing practices or inadequate feedback can lead students to request extra support from the teaching staff, meaning that their autonomy is reduced and that valuable teacher time is consumed. Consequently, this undermines two key benefits of employing AATs: fostering student autonomy and optimizing teacher resource allocation. Furthermore, certain testing patterns can result in assessments which are too strict and even unfair, either by letting students receive better grades than they should – for example, by considering hardcoded solutions as correct – or by penalizing them too much – for example, by not informing students when their code fails due to minor formatting issues.

From our experience, we have derived a set of recommendations which we share with new teaching assistants who join our courses and have to prepare new auto-graded assignments.

In this paper, we share those lessons with the Computer Science Education (CSE) community. We believe this information might be useful to other teachers and instructors who wish to use start using or already use AATs.

Finally, the recent advent of Large Language Models (LLMs) – artificial intelligence tools with capacity to generate computer code from natural language descriptions – such as OpenAI’s GPT and Google’s Bard has generated some controversy in the CSE community, due to the risk of producing graduates who are weak in fundamental computing skills such as computational thinking and program design [11]. While LLMs pose certain risks, they also offer new opportunities for the CSE community. We explore how integrating AATs with LLMs can generate beneficial possibilities.

This paper makes the following contributions:

- Presents a list of recommendations for defining unit tests for automatic assessment, aimed at enhancing student autonomy and grading fairness;
- Presents adaptations of those recommendations for specific assessment types;
- Discusses how Large Language Models can play a role in the future development of AATs.

This paper is organized as follows: first, in Section 2, we review relevant papers related with automatic assessment, feedback and LLMs. In Section 3 we present 1) a categorization of assessment types typically found in computer programming courses, 2) some AAT concepts, and, 3) some Unit Testing concepts. Section 4, presents our main contribution: recommendations for test-case creation and feedback definition. Section 5, discusses the future of AATs in light of the recent advent of LLMs. In Section 6 we address some threats to the validity of this paper. Finally, in Section 7, we finish our paper by drawing some conclusions.

## 2 Related work

**Automated Assessment.** AATs have been used since at least 1960 [13]. These tools evaluate students' code solutions to determine if they satisfy the requirements of the assignment, giving students more autonomy and reducing teacher workload [13]. This is usually done by automating the execution of students' code in a restricted environment and validating its correctness by executing unit tests, performing output matching, or by other techniques [9].

Research on this area is varied, with many studies presenting tools, their advantages, limitations and impact on students' learning [5], amongst other topics [9, 13]. Some of these tools attempt to simulate professional software development work environments promoting that students work using Integrated Development Environments ([3, 2]) and even complementary tools such as Git (e.g. [8, 3]), while others introduce more virtual work environments which are usually implemented using web-based solutions (e.g. [4, 16]).

**AAT cheating patterns.** Research has also delved into students' ability to work around automatically graded assessments and get successful scores without performing the assignment as prescribed by the teachers. The authors of [10] analysed students' code submissions in 2 AAT-based courses, identifying and categorizing cheating attempts into 4 distinct patterns:

- **Overfitting:** Students' code is hardcoded to match expected output; ineffective beyond the specific assignment's tests. Overfitting corresponded to 63% of the cheating attempts.
- **Problem evasion:** Students' code fails to meet non-functional requirements (e.g. the use of recursion). It accounted for 30% of the cheating attempts.
- **Redirection:** Students call the reference solution from their own code. Redirection corresponded to 6% of the cheating attempts;
- **Injection:** Malicious code is injected to change the assignment's validation process. This pattern corresponded to 1% of the cheating attempts.

Some of these issues, particularly Redirection and Injection, are highly dependent on the AAT's architecture. For instance, AATs that isolate the reference solution from the students' code, such as [3], indirectly prevent Redirection. However, both Overfitting and Problem evasion are cheating patterns which do not directly exploit AATs' weaknesses.

## 3:4 Seven Years Later: Lessons Learned in Automated Assessment

**Feedback.** Feedback is considered a facilitator of learning and performance [15]. In a literature review focused on feedback, Shute defined formative feedback as *information communicated to the learner that is intended to modify his or her thinking or behaviour to improve learning* [15]. Formative feedback can be presented to the learner using information (e.g. verification of response accuracy, explanation of the correct answer, hints) and can be administered at various times during the learning process (e.g. immediately following an answer, after some time has elapsed) [15]. This research also offered guidelines for creating formative feedback. In a recent literature review focused on formative feedback for computer science students [17], the authors concluded that, to better assist novice programmers in learning programming, formative assessments must improve the presentation of error messages. Finally, in a study evaluating the use of GPT-3.5 for generating hints for problems detected by an AAT [14], students were divided into two groups – one receiving GPT-based hints and the other only receiving regular AAT feedback. The usefulness of the GPT hints was assessed using a 5-point Likert scale; 46% of students rated them highly (4 or 5), while 19% gave low ratings (1 or 2). The study found that students with access to GPT-based hints tended to use the AAT’s default feedback less. However, the findings also suggest some potential over-reliance on the GPT-generated feedback.

**In summary,** despite extensive research in this field, we are not aware of any published studies providing guidelines or recommendations for creating test functions/cases and determining the appropriate level of feedback to support programming students in their learning.

### 3 Definitions

#### 3.1 Assessment types

Our experience with automated assessment has led us to conclude that different assessment (or evaluation) types require different approaches, due to their specificities (e.g. whether the evaluation is time-bound or not).

This section presents the 4 evaluation types for which we recommend that different levels of feedback are considered and provided. These definitions should help teachers bridge our evaluation types with their own, and thus be able to more effectively apply our lessons and recommendations.

**In-class/homework exercises:** These are usually small exercises in which students have to implement code to solve small problems (e.g. 20-100 LOC) using specific techniques (e.g. iteration, recursion, etc). These are usually formative, although they might have some contribution to the students’ grades. During said exercises, students can get support from both teachers as well as their colleagues.

**Mini-test:** Small exercises to be done individually and independently (e.g. without support). These exercises are time-bound (e.g. maximum one hour) and count towards the students’ final grade.

**Project:** Mid-to-large scale exercises which take weeks or even months to solve. These usually have a larger weight in the students’ final grades.

**Defense:** A proctored evaluation where students must make changes to their assignment’s code, in order to demonstrate knowledge of it. This is usually done for projects and other evaluations with significant impact in the students’ grades, in order for teachers to have more confidence that students actually authored the delivered code instead of copying or plagiarising. For group projects, it has the added benefit of adjusting the grades given to different group elements.

## 3.2 Test function types

Usually AATs inform students about their results. However, some AATs allow the definition of hidden tests, which are tests whose existence and result is not disclosed.

**Public tests:** Test functions whose results are reported to the student (e.g. correct or incorrect) who possibly also receives messages with extra feedback.

**Private or hidden tests:** Test functions which are executed without the student receiving any feedback. Hidden tests are a common strategy to counter the usage of overfitted solutions: teachers prepare a public test for a function and then prepare a private test for the same function, but with different test cases. Since students will not even know that the test exists, they will not be able to derive an overfitted solution that passes both the public and the hidden tests. Hidden tests also have the potential for better distinguishing between students' grades, since students who test their code locally with more cases than the ones made available in the public test function might end up also passing the hidden tests. However, this will probably depend on the cases which are actually been tested and whether they are corner and/or otherwise complex cases.

## 3.3 Unit Testing frameworks

Unit Testing frameworks typically supply programmers with assertions: functions which verify if a certain condition is being respected. Some notable examples are: “assertEquals()”, “assertTrue()”, and “assertFalse()”. They also typically have a “fail()” function, which can be used when the programmer wants to test a condition without using one of the default assertion functions.

These assertion functions usually receive an optional “message” parameter, which can be used to customize the message that is displayed when the assertion fails. In professional software development practice, this parameter is usually not very relevant, because programmers have access to the test code and can observe the assertion's implementation details. However, it gains importance when developing tests for AATs, since students will usually not have access to the assignment's test code.

Several Unit Testing frameworks exist for a variety of programming languages. The most well-known example for Java is *JUnit* [6]. Another example is Python's *unittest*<sup>3</sup> library.

## 4 Lessons Learned: recommendations

This section presents our recommendations. The examples are given in Java, using *JUnit* 5, but they can easily be adapted to other languages and frameworks, such as Python.

### 4.1 Generic recommendations

#### 4.1.1 Tackling problem evasion

In order to detect cases where students try to evade the prescribed problem (e.g. recursion), we recommend that static analysis tools are employed in the assessment process. An example tool which can be used to do this for Java code is the Checkstyle plugin<sup>4</sup> which, among other functionality, allows teachers to define forbidden keywords (e.g. “for”, “while”) and signals their usage. This enables teachers to inspect and/or penalize solutions which use those keywords.

<sup>3</sup> <https://docs.python.org/3/library/unittest.html>

<sup>4</sup> <https://checkstyle.org/>

Another useful technique is Reflection [12]. This technique allows certain non-functional requirements, such as the need for a class to be abstract, to be validated using unit tests.

#### 4.1.2 Test construction

Before we delve into our recommendations for test case construction, it should be noted that, in some scenarios, it might be necessary for the exercises to be changed in order for them to be properly evaluated in an automatic fashion using unit tests. This means that test functions influence API design, similarly to what happens in test driven development. In order to apply some of our recommendations, the tested functions should always receive at least one parameter, and the more parameters, the easier it will be to create rich test-cases. Consider this when evaluating our first 2 recommendations: they might seem somewhat contradictory, but they are achievable by adjusting the exercises' APIs.

##### At least 2 independent test functions for each API function

We recommend creating at least two independent test functions for each of the exercise's API. As an example, consider this exercise: *Implement a recursive function that compares two char arrays through an extra argument representing the initial position (assume a1 and a2 have equal length):* `static boolean equalArrays(char[] a1, char[] a2, int initialPos)`

You can break this problem down into different test functions, considering different input cases, such as 1) empty or null arrays, 2) initial position greater than the length of the array, 3) arrays with just one element, and, 4) arrays with 2 or more elements. By having multiple small test methods for each function, you allow gradual exercise completion with tests of increasing difficulty (an essential ingredient of gamification). Not only does this increase the student's motivation, but it also allows for more fine-grained grading (i.e. a student who is able to implement the base case but not the recursive case doesn't have zero). Also note that each of these test-methods can further help the student by having a name which hints at the case being tested (e.g. `testEqualArraysSingleElement()`).

##### At least 2 sub-cases in each test function

Each test should have at least 2 sub-cases for the same function, with different expected return types, in order to prevent hardcoded/overfitted solutions from being considered correct, even if only partially. This should be implemented by having 2 different assertions with different return values. The most obvious example for this appears when evaluating boolean functions: if the test function only has cases for when the expected value is "true" (as in Listing 3), then a solution which always returns "true" is going to pass that test.

■ **Listing 3** Test function which validates a boolean function using a single assertion. This test will pass if the student's `isOpen()` function returns an hardcoded "true".

```

1 @Test
2 public void testDoorOpening() {
3     Door door = new Door();
4     door.open();
5     assertTrue(door.isOpen(), "isOpen() incorrectly returned false");
6 }

```

## On Feedback

Since we want to help students to solve the exercises, assertions without a specific message should be avoided (consistent with previous recommendations on feedback generation [15]). As such, the minimum feedback should, at least, indicate the name of the function returning the incorrect result (e.g. “getSalary() returned the wrong value” or, at least, “getSalary()”). This recommendation might be slightly relaxed if we are testing a function with a very specific return format which will make it obvious for the students which is the tested function. For example, when assessing an object’s “toString()” method, if its output format significantly differs from that of other objects, this approach may be appropriate.

However, just mentioning the function name might not be sufficiently helpful. For example, consider the assertion in Listing 4:

■ **Listing 4** In certain cases, just referring the function name is insufficient and might cause confusion.

```
1 List<String> movieTitles = Main.getTitles();
2 boolean found = movieTitles.contains("The Matrix");
3 assertTrue(found, "getTitles()");
```

If the title “The Matrix” is not included in the list, the respective assertion will simply yield “AssertionFailedError: getTitles() => Expected: true Actual: false”. This will let the student know that something is wrong with the function “getTitles()”, but, unless the problem is very obvious, debugging will be hard since due to lack of clues. Also, we have seen situations where this pattern has led some students to confusion, due to the expected/actual result indicating “true” and “false” when the “getTitles()” function is not supposed to return a boolean. It would be more helpful to let students know which value is missing so that they can direct their debugging efforts. Refer to Listing 5 for an alternative implementation which contains more actionable feedback:

■ **Listing 5** A test with more actionable feedback than the one in Listing 4.

```
1 List<String> movieTitles = Main.getTitles();
2 String title = "The Matrix";
3 boolean found = movieTitles.contains(title);
4 assertTrue(found, "getTitles() missing title: " + title);
```

Of course, giving students this extra piece of information could lead to partial or full overfitting, since the student could insert the “The Matrix” String in the returned list to circumvent their original error. In fact, students’ ability to produce overfitted solutions is highly dependent on the information which is available to them, both via the assignment’s description as well as via the employed test cases. To better understand this idea, consider the example unit test presented in Listing 1. That test can be easily bypassed using the solution shown in Listing 6:

■ **Listing 6** Overfitted solution that passes the test defined in Listing 1. It would be easy for a student to hardcode a solution for that test after seeing its feedback.

```
1 static long sum(int n1, int 2) {
2     if(n1 == 1) {
3         return 5; // first assertion indicates 1 + 4
4     }
5     else if(n1 == 8) {
6         return 10; // second assertion indicates 8 + 2
7     }
8     return 0;
9 }
```

As such, it is necessary to find a balance between the level of feedback which is given to students: on one hand, too much feedback might lead to overfitted solutions, since, if students know all expected input/output pairs, then they can prepare a function which returns the correct result but is not generic. On the other hand, too little feedback will leave students lost and unable to progress. This is particularly important when dealing with API functions such as *JUnit*'s “`assertTrue()`” and “`assertFalse()`”, which provide no clue other than “`AssertionFailedError: Expected: true Actual: false`”. This problem is made even worse in courses where the tests might be calling multiple functions, which commonly happens in OOP exercises. In those cases, such reduced feedback will likely make students wonder which of their functions has the problem.

However, with a varied number of test cases per test function (as previously recommended), it will be hard for students to implement an overfitted version which passes all the sub-cases. Furthermore, teachers can make testing functions which give a lot of detail in the first few cases, and little to no detail in the final cases ([15]). See Listing 7 for an example.

■ **Listing 7** Test function with a progressive reduction of feedback level.

```

1  @Test
2  public void testSumArray() {
3      String suffix = "returned an incorrect value.";
4
5      // feedback includes the arguments (vulnerable to overfitting)
6      int[] array1 = new int[]{1, 4};
7      assertEquals(5, Main.sum(array1), "sum({1, 4}) " + suffix);
8
9      // feedback includes the arguments (vulnerable to overfitting)
10     int[] array2 = new int[]{2, 2, 2};
11     assertEquals(6, Main.sum(array2), "sum({2, 2, 2}) " + suffix);
12
13     // feedback doesn't include the arguments, only a hint
14     String suffix2 = suffix + "Consider arrays w/ negative numbers.";
15     int[] array3 = new int[]{-1, -2, -3};
16     assertEquals(-6, Main.sum(array3), "sum(...) " + suffix2);
17
18     // feedback only includes the function name
19     int[] array4 = new int[]{1, 2, 3, 4, -1};
20     assertEquals(9, Main.sum(array4), "sum(...) " + suffix);
21 }

```

In some situations, besides the name of the function and some hints ([15]) with regards to the expected values, the minimum feedback should also indicate relevant particularities of what is being tested. For example, imagine that we are testing a function called “`int[] getStatistics()`” which returns an integer array where each element has a different semantics: the first element is the input array’s minimum value, while the second element is the input array’s maximum value. If we provide students with simply:

```
assertEquals(42, Main.getStatistics()[1], "getStatistics()");
```

debugging will be hard due to lack of information. It can also cause interpretation issues, since the student might be confused when seeing a message similar to “`getStatistics() expected: 42 but was: 99`” since “`getStatistics()`” returns an “`int[]`” and not a single “`int`”. In this example, it would be preferable to define the test’s feedback in order to let the student know which of the array’s positions has the wrong value, as follows:

```
assertEquals(42, Main.getStatistics()[1], "getStatistics()[1]");
```



This will give the student extra clues with regards to where to start analysing the bug.

Finally, the amount of feedback also depends on the exercise type. This will be further expanded later in this paper.

### On hidden tests

Avoid using hidden tests to validate things such as String contents. It is easy for students to make minor mistakes, such as having an incorrect capitalization or spelling mistake. Validating such behaviours only in a hidden test might be too penalizing. Furthermore, although hidden tests can be an interesting tool, they can also be the source of problems: since their results are not visible to the students, eventual errors are likely to go unnoticed. An option to mitigate this issue is to use an AAT which is able to indicate how many students are passing each test [3], and, if no student passes a test or only a small minority of students pass it, then maybe the teaching staff should review it, since it might be incorrect.

### Assert-early and often

Consider performing one or more assertions after each action in the test, instead of performing multiple assertions after executing a number of actions. This is particularly helpful when validating OOP assignments, where an object's state changes with each action, meaning that an error on the first action might cause problems in the following actions. If the first assert is done after multiple actions, the resulting feedback might induce the student in error. Listing 8 presents an example of this strategy.

■ **Listing 8** Example of the assert-early and often strategy. After each action, a number of assertions is performed, in order to point to the bug as early as possible.

```
1 Door door = new Door();
2
3 // perform an action
4 door.open();
5 // assert that the object's state is 'open' after the first action
6 assertTrue(door.isOpen(), "isOpen() incorrectly returned false");
7 // also validate the object's toString()
8 assertEquals("This door is open!", door.toString());
9
10 // perform another action
11 door.close();
12 // assert that the object's state is 'closed' after the second action
13 assertFalse(door.isOpen(), "isOpen() incorrectly returned true");
14 // also validate the object's toString()
15 assertEquals("This door is closed!", door.toString());
```

### Null references

Be careful when creating test cases which might result in hard to debug errors such as “NullPointerException”. Consider using “assertNotNull()” to let students know where their code is returning unexpected null references. See Listing 9 for an example.

## 3:10 Seven Years Later: Lessons Learned in Automated Assessment

■ **Listing 9** Gracefully handling potential null references to provide useful feedback.

```
1 // unprotected approach, which will result in a hard to debug
2 // NullPointerException
3 Actor actor1 = Main.getActorByID(1000);
4 assertEquals("Keanu Reeves", actor1.getName(), "Actor getName()");
5
6 // protected approach, which will let the student know that
7 // an unexpected null reference was returned
8 Actor actor2 = Main.getActorByID(1000);
9 assertNotNull(actor2, "getObject(1000) returned null");
10 assertEquals("Keanu Reeves", actor2.getName(), "Actor getName()");
```

### Test-function names

Although the industry norms and conventions for test code do not recommend prefixing test functions with “test\_” nor having a number for each test, we believe that these two practices are actually helpful in educational contexts. Including said prefix has the advantage of facilitating the interpretation of the tests’ stack-traces (when available) since it makes it obvious which of the functions in the stack is the test. Furthermore, numbering the test makes it somewhat easier to communicate with the student.

### The order of the tests matters

Even though, according to the best practices, the tests should be independent of each other, students tend to tackle each failed test by the order it shows up in the report provided by the AAT. As such, consider ordering the tests by difficulty level.

### Progressive disclosure

Consider activating certain advanced tests only after students pass simpler tests. This is not directly accomplishable by *JUnit* (since it goes against the philosophy of independent tests) but it is easily implemented using static variables (see Listing 10). Important: Students should know that these tests exist and what is needed to “unlock” them (e.g. “This test will only be executed after you pass the tests based on the small input files”). This rule also interacts with the previous one: the feedback for more complex tests should appear after the feedback for the simpler tests.

■ **Listing 10** Progressive disclosure: this test is only executed after the student passes at least 4 simple tests.

```
1 @Test
2 public void advancedTest() {
3     if (SimpleTests.testsOK < 4) {
4         fail("This test will be unlocked after passing 4+ simple tests");
5     }
6     // ... actual test
7 }
```

### Tests involving collections

Be careful with “OutOfBounds” runtime errors: check the length of the collection before accessing individual elements. However, don’t just check the total length of the collection upfront, since it will be difficult for students to understand the problem when they have insufficient or excessive data. Instead, first validate a few elements in order to allow students to debug missing values. See Listing 11 for examples of both approaches.

■ **Listing 11** Testing collections' contents: its better to validate a few examples before asserting the full list size.

```
1 List ids = Main.calculateIds();
2
3 // this is difficult to debug
4 assertEquals(347, ids.size(), "calculateIds() returned wrong number of elements");
5
6 // this is better
7 int size = ids.size();
8 assertTrue(size >= 2, "calculateIds() should have returned at least 2 elements, but
9   returned " + size);
10 assertEquals(3, ids[0], "calculateIds()[0]");
11 assertEquals(5, ids[1], "calculateIds()[1]");
12 assertTrue(size >= 5, "calculateIds() should have returned at least 5 elements");
13 assertEquals(6, ids[4], "calculateIds()[4]");
```

### Include a timeout on each test

In order to detect infinite loops and/or implementations with inappropriate time complexity (i.e. big-O), teachers should consider defining timeouts for some or all of the test functions.

## 4.2 Specific recommendations per assessment type

As we've seen before, in order to allow students to act and solve a problem reported by the AAT, the respective tests should supply the student with some level of information. We argue that the ideal level of information will be different depending on the assessment type.

### In-class/Homework exercises

Due to the formative goals of these exercises, we consider it important to provide students with feedback which allows them to fully master the material. As such, all test functions should be public and all, or at least the majority, of assertions should have significant feedback.

### Mini-tests

In such assignments, we recommend that some feedback is given, but at least one of the asserts should give very little feedback, in order to reduce the likelihood of *overfitted* solutions. For example, at least the last assertion of each test function should not give any hints in terms of input and expected output. All test functions should be public, since it is important for students to have a real notion of how they are doing.

### Projects

Use hidden tests to make it harder for *overfitted* solutions to get good grades. Define a subset of the public tests as “mandatory”. These tests will define the minimum mandatory functionality which a student must implement in order to get a passing grade, also promoting that the approved projects have a minimum level of quality. Furthermore, if you employ project defenses, having mandatory tests will aid in avoiding certain issues (refer to the next paragraph for details). Hidden tests should not be mandatory, as students are unaware of their existence and cannot take action to address or rectify any related issues.

## Defenses

The rules presented for mini-tests also apply to project defenses, since both are time-bound evaluations where students tend to become stressed. However, a few additional recommendations make sense when defining tests for project defenses. If possible, build the defenses around the project’s “mandatory functionality” in order to prevent penalizing students whose projects have bugs that were not detected originally. If “mandatory” tests were not defined, teachers should avoid testing scenarios which were only tested in the project’s hidden tests. These two recommendations aim at reducing the chance of penalizing students whose projects have pre-existing bugs that might condition their performance in the defense. Furthermore, for similar reasons, we recommend using adapted versions of the project’s public tests in the defense’s tests (e.g. same input file, similar actions, and so on). Teachers should also be careful with interdependence between the test cases. For example, if the defense’s assignment asks students to change the “toString()” function, then it might be dangerous to use the “toString()” as a criteria to determine the correctness of other defense tests. Finally, common pitfalls should be identified and avoided. For example, in projects involving people’s names, some students only support two-named individuals and overlook those with multiple names, like “Samuel L. Jackson” versus “Samuel Jackson.” It’s important to recognize and steer clear of these issues in project defenses, as students often fail to address them despite detailed feedback.

### 4.3 Other recommendations

Sometimes students complain that a test is incorrectly failing, justifying their position with something along the lines of “in my computer, it works”. As such, we find it important to educate students on how their code will be tested.

One way of doing this is to teach them how to programmatically test their own code. Even so, some students might still struggle with transferring this skill to the interpretation of teachers’ tests, due to lack of testing experience and/or other issues. To help with this, consider giving students access to the code of one of the assignment’s tests, in order to allow them to reflect on how testing is being performed. This might have the added benefit of students making their own adaptations of the sample test, thus better testing their code.

Finally, we also suggest a strategy of promoting that students write their own tests when they ask for help. In response to help requests, we often prompt students to demonstrate a unit test that evaluates a specific aspect, akin to our AAT’s test. Although this usually requires some interactions until the student produces a test which mimics the actual problem, it has the advantage of forcing students to enter a testing-oriented mindset.

## 5 The Future of Automatic Assessment

Although LLMs have been received somewhat controversially by the CSE community [11], we expect CS educators to adopt these technologies in their teaching practices sooner or later. More concretely, we believe LLMs will be used to improve AATs and/or assessment practices [1].

Some possible research avenues are: 1) using LLMs to automate the generation of tests ([11]) that respect the recommendations presented in this paper; 2) integrating LLMs and AATs to simplify the process of providing hints for failed tests, similarly to [14], but with guardrail mechanisms to prevent LLM over-reliance; 3) employing LLMs to generate customized feedback that considers both the student’s code as well as the issues identified

by the AAT; 4) integrating LLMs into AAT's evaluation schemes ([7]) to detect issues such as overfitting and problem evasion; 5) replacing AATs with LLMs entirely. Note that developments in terms of improving and/or complementing AAT feedback (i.e. #2 and #3) might reduce the need for educators to follow recommendations such as the ones presented in this paper. With regards to #5, we find it unlikely, at least in the near future, since current state-of-the-art LLMs have limitations (namely, hallucinations<sup>5</sup>), and still require human supervision [7]. The need for said supervision would potentially negate one of the biggest advantages of automated assessment: reduction of teacher workload.

## 6 Limitations

The recommendations in this paper are based on our multi-year experience across various courses but have not been statistically validated. As such, it is possible that they are somewhat biased. To mitigate this issue, we have reviewed our recommendations in relation to earlier studies on feedback, such as [15], and confirmed their alignment with prior recommendations.

## 7 Conclusion

This paper presented a set of recommendations for the definition of unit tests to automatically verify students' work and provide students with formative feedback which allows them to learn and/or act on their mistakes. We believe that following these recommendations has helped us help our students in their learning processes. They also helped us avoid certain mistakes which would have otherwise consumed our time or penalized students excessively.

---

## References

- 1 Bruno Pereira Cipriano. Towards the Integration of Large Language Models in an Object-Oriented Programming Course. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 2*, ITiCSE 2024, pages 832–833, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649405.3659473.
- 2 Bruno Pereira Cipriano, Bernardo Baltazar, Nuno Fachada, Athanasios Vourvopoulos, and Pedro Alves. Bridging the Gap between Project-Oriented and Exercise-Oriented Automatic Assessment Tools. *Computers*, 13(7), 2024. doi:10.3390/computers13070162.
- 3 Bruno Pereira Cipriano, Nuno Fachada, and Pedro Alves. Drop project: An automatic assessment tool for programming assignments. *SoftwareX*, 18:101079, 2022.
- 4 Stephen H Edwards and Krishnan Panamalai Murali. CodeWorkout: Short Programming Exercises with Built-in Data Collection. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education*, pages 188–193, 2017.
- 5 Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. Five Years with Kattis — Using an Automated Assessment System in Teaching. In *2011 Frontiers in education conference (FIE)*, pages T3J–1. IEEE, 2011.
- 6 Boni Garcia. *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing Ltd, 2017.
- 7 Sklyer Grandel, Douglas C Schmidt, and Kevin Leach. Applying Large Language Models to Enhance the Assessment of Parallel Functional Programming Assignments. In *Proceedings of the 2024 International Workshop on Large Language Models for Code*, pages 1–9, 2024.

---

<sup>5</sup> Occasions in which LLMs produce outputs which are factually incorrect.

- 8 Sarah Heckman and Jason King. Developing Software Engineering Skills using Real Tools for Automated Grading. In *Proceedings of the 49th ACM technical symposium on computer science education*, pages 794–799, 2018.
- 9 Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93, 2010.
- 10 Nane Kratzke. Smart Like a Fox: How Clever Students Trick Dumb Automated Programming Assignment Assessment Systems. In *CSEDU (2)*, pages 15–26, 2019.
- 11 Sam Lau and Philip Guo. From “Ban it till we understand it” to “Resistance is futile”: How university programming instructors plan to adapt as more students use AI code generation and explanation tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*, pages 106–121, 2023.
- 12 Yue Li, Tian Tan, and Jingling Xue. Understanding and Analyzing Java Reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(2):1–50, 2019.
- 13 José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, 22(3):1–40, 2022.
- 14 Maciej Pankiewicz and Ryan S Baker. Large Language Models (GPT) for automating feedback on programming assignments. *arXiv preprint*, 2023. [arXiv:2307.00150](https://arxiv.org/abs/2307.00150).
- 15 Valerie J Shute. Focus on Formative Feedback. *Review of educational research*, 78(1):153–189, 2008.
- 16 Igor Škorić, Tihomir Orehovački, and Marina Ivašić-Kos. Exploring the Acceptance of the Web-based Coding Tool in an Introductory Programming course: A pilot Study. In *Human Interaction, Emerging Technologies and Future Applications III: Proceedings of the 3rd International Conference on Human Interaction and Emerging Technologies: Future Applications (IHET 2020), August 27-29, 2020, Paris, France*, pages 42–48. Springer, 2021.
- 17 Jagadeeswaran Thangaraj, Monica Ward, and Fiona O’Riordan. A Systematic Review of Formative Assessment to Support Students Learning Computer Programming. In *4th International Computer Programming Education Conference (ICPEC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.