Beyond Functionality: Automating Algorithm Design Evaluation in Introductory Programming Courses

Caio Oliveira, Leandra Silva and João Brunet
Department of Systems and Computing, Federal University of Campina Grande, Campina Grande, Brazil
{*caio.oliveira, leandra.silva*}*@ccc.ufcg.edu.br, joao.arthur@computacao.ufcg.edu.br*

Keywords: Design Test, Introductory Programming Courses, Tests, Feedback, Assessment.

Abstract: Introductory programming courses often enroll large numbers of students, making individualized feedback challenging. Automated grading tools are widely used, yet they typically focus on functional correctness rather than the structural aspects of algorithms—such as the use of appropriate functions or data structures. This limitation can lead students to create solutions that, while correct in functionality, contain structural flaws that might hinder their learning. To address this gap, we applied the concept of design tests to automatically detect structural issues in students' algorithms. Our tool, PyDesignWizard, provides an API leveraging Python's Abstract Syntax Tree (AST) to streamline the creation of design tests. We quantitatively evaluated this approach on 1,714 student programs from an introductory course at the Federal University of Campina Grande, achieving 98.6% accuracy, a 0.99 true negative rate, 77% precision, and 84% recall. A qualitative evaluation was also conducted through interviews with 16 programming educators using the Think Aloud Protocol. The results indicated a high level of understanding and positive feedback: 15 educators grasped the concept quickly, and 8 highlighted the educational benefits of identifying design issues. In summary, our approach empowers educators to write structural tests for automated feedback, enhancing the quality of grading in programming education.

1 INTRODUCTION

The teaching of programming has become increasingly important over the years, driven by technological advancements and the growing interest in Information Technology. This growth has resulted in larger classes in introductory programming courses, significantly increasing the number of exercises and exams that instructors need to assess, provide feedback, and grade (Watson and Li, 2014). For example, in the context that this work took place, the Introduction to Programming course at the Federal University of Campina Grande (UFCG), a class typically consists of 90 students per semester. The course adopts a practice-focused approach, where students are required to regularly submit exercises. On average, each class generates around 270 exercise submissions, which the instructor must assess, provide feedback on, and grade. This significant workload highlights the challenges faced by instructors in scaling assessment practices while ensuring timely and effective feedback.

Studies such as those by Leite (Leite, 2015) and Papastergiou (Papastergiou, 2009) suggest that increasing the frequency of programming exercises can help improve students' skills, but this also increases the workload for instructors. Automated grading tools, as mentioned by Janzen (Janzen and others, 2013) and Singh (Singh et al., 2013), are often used to address this issue. Instructors frequently use Online Judges (Juiz and others, 2014), where students' submissions are automatically graded based on tests that primarily focus on functional correctness, possibly neglecting important aspects related to code structure and algorithm design. In summary, the solutions tend to focus on testing only 'what' the students' code does, rather than 'how' they do it. While there are studies exploring the use of static analysis in programming education (Truong et al., 2004), these approaches primarily focus on code quality rather than addressing algorithm design (Araujo et al., 2016).

In addition to functional correctness, it is essential to evaluate the design of algorithms. Design refers to the elements in the algorithm, and how the algorithm is structured and organized in code. Due to the limited focus in the literature on how to automatically verify the algorithm design, and because Online Judges focus primarily on functional aspects, many students end up implementing the exercises violating the ex-

Oliveira, C., Silva, L. and Brunet, J.
Beyond Functionality: Automating Algorithm Design Evaluation in Introductory Programming Courses.
DOI: 10.5220/0013242000003932

In *Proceedings of the 17th International Conference on Computer Supported Education (CSEDU 2025) - Volume 2*, pages 516-525
ISBN: 978-989-758-746-7; ISSN: 2184-5026

pected design and hindering the development of logical reasoning skills. For example, consider that an instructor asks students to implement the Merge Sort algorithm in Python (Foundation, 2023), but the student chooses to implement the Bubble Sort algorithm instead. Although the solution is functionally correct, that is, it sorts a sequence, the student does not meet the instructor's design expectations. In some cases, the problem is even worse, when students end up using pre-built libraries or functions (sort() in this example) that do all the work that they were supposed to do.

To address this gap, we propose an automated approach for inspecting algorithm design, based on static code analysis, with a focus on the educational context. Our tool, *PyDesignWizard*, provides an API that allows the creation of design tests that verify the structure of algorithms, beyond functional correctness. Unlike traditional tests, our approach covers everything from parameters and variables to detecting loops and recursion, enabling a deeper evaluation of syntactic and semantic conformity. This allows instructors to automatically check aspects of the design of the code, such as control flow structures usage, recursion depth, complexity, code modularity, among others.

The *PyDesignWizard* API was built based on Python's abstract syntax tree (AST)(Beazley, 2012)(van Rossum, 2009), facilitating code reading and manipulation. With this tool, instructors can create their own test suites to evaluate both the functionality and design of students' algorithms, ensuring the originality of their solutions and optimizing grading time.

In order to evaluate the ability of our approach to detect design issues and its practical applicability for instructors, we conducted our research guided by the following research questions:

- **RQ1.** Can the structure of students' solutions be effectively assessed through design tests?
- **RQ2.** Do educators perceive the concept of design tests as intuitive and easy to grasp?
- **RQ3.** Does a tool based on design tests offer tangible benefits in educational contexts?

To validate our tool and answer these questions, we conducted two distinct studies. The first study evaluated the accuracy of algorithm detection in 1714 programs developed by 90 students in the Introduction to Programming course at the Federal University of Campina Grande, one of the top 5 largest Computer Science courses in Brazil. In order to understand how instructors perceive our approach, we have conducted a qualitative study interviewing 16 instructors from different universities. We have applied the Think Aloud Protocol (Ericsson and Simon, 1993a) to gather and analyze data from the interviews.

The results indicate that the tool was effective in detecting algorithm details and that design tests reduced the incidence of false negatives. Specifically, the test achieved a 98.6% accuracy, with a precision of 77% and a recall of 84% in detecting sorting algorithms. In the qualitative assessments, instructors found the design tests intuitive and recognized the educational potential of the tool in the context of programming education. Eight interviewees highlighted the tool's usefulness in verifying the structure of students' code, while four stated they would use it in their courses. Another four suggested that the tool could be incorporated into courses on testing and software architecture, complementing existing practices such as unit testing. Only two interviewees did not find a clear use for the tool in their educational context. Overall, the feedback suggests that the tool can serve as an ally in improving the quality of programming education by encouraging best practices in algorithm design and code structure.

This paper is structured as follows. First we provide the background for this work, discussing the concept of Design Tests in Section 2. Then, in Section 3, we describe how we apply the concept of Design Test for educational purposes, along with the details of the tool we built to support the implementation of design tests for Python programs. In Section 4 we present both studies we conducted: a quantitative evaluation to demonstrate our tool's ability to identify structural design elements in student code with high accuracy and precision, and a qualitative evaluation to gather feedback from instructors on the usability and potential educational benefits of the tool. Finally, in Section 6, we summarize our findings, discuss potential threats to validity, and suggest directions for future research, including expanding the tool's applicability to other programming paradigms and educational contexts."

2 BACKGROUND: DESIGN TESTS

In a previous work (Brunet et al., 2009), we introduced the concept of design tests, an innovative approach to test software aimed at verifying whether a software system's implementation adheres to specific architectural rules or design decisions. While traditional functional tests focus on ensuring that a program delivers correct outputs in response to given inputs, design tests go beyond this by checking how

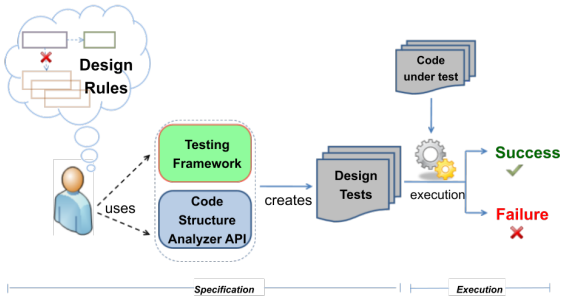

Figure 1: Design Test Overview.

the system has been implemented according to predefined architectural guidelines. This method helps to ensure that design decisions are consistently applied, particularly in teams where developers may change frequently, reducing the likelihood of missed or misunderstood design aspects. By incorporating design tests into the daily testing routine, mismatches between design and implementation can be detected early, improving overall software quality.

To support the use of design tests for architectural rules, we developed DesignWizard, an API integrated with JUnit (JUnit Team, 2023) for Java implementations. This tool allows developers to express design rules and decisions in a programming language familiar to them, making the documentation of design constraints both executable and easy to maintain. Figure 1 illustrates an overview of the design tests approach. In this method, a developer creates a design test to automate the verification of design rules. The two key components of this approach are the code structure analyzer API, which extracts metadata about the code and exposes this information, and the testing framework, which provides assertion routines, automates test execution, and reports the results.

For the sake of clarity, let us analyze a concrete example. The design test below (Listing 1) illustrates this: a design rule dictates that classes in the `dao` package should only be accessed by classes within the `controller` or `dao` packages. This rule is verified through a design test that checks all method calls to ensure compliance.

Listing 1: Design Test pseudo-code for communication rule between dao and controller package.

```
1 daoPackage = myapp.dao
2 controllerPackage = myapp.controller
3 FOR each class in daoPackage DO
4   callers = class.getCallers()
5   FOR each caller in callers DO
6     assert caller in controllerPackage ||
7         caller in daoPackage
```

This design test is constructed by first identifying the target packages and then checking the callers of each class in the `dao` package. Assertions are made to verify that these callers belong to the `controller` or `dao` packages. The test relies on static analysis provided by DesignWizard to retrieve structural information, and the JUnit framework is used to automate the process.

In the next section, we describe our proposal to adapt and apply the concept of design tests to the context of programming education. We have adapted the concept of design tests to the level of algorithms in order to allow instructors to write and assess more fine-grained rules related to the usage of loops, variables, conditionals, among others.

3 DESIGN TESTS FOR EDUCATION

Context. In the context of education, the scenario for instructors is similar to developers: there are a plethora of tools and mechanisms to verify whether students' programs are functionally correct, that is, whether their solutions do what they are supposed to do, but, to the extent of our knowledge, there is no approach to check the internal aspects (the design) of the algorithms they produce using the same language and test infrastructure of the target program. In this paper, we propose, implement, and evaluate the tooling needed to apply the concept of design tests to the context of education.

Design at Algorithm Level. As one may note, the concept of design is broad and applied to different abstraction levels. As described in Section 2, design tests were proposed in the context of high-level abstractions, for example, modules, packages, and subsystems. In the context of introductory programming courses, on the other hand, the instructor's concerns are at a different level, but still related to the structure; hence, the design. These concerns are related, but not limited to:

- the elements students use to implement the program, such as conditionals, loops, variables, data types, and operators;
- the choice and manipulation of data structures; and
- the usage of specific functions, classes, and modules.

The Problem. Let us take as a concrete example the Introduction to Programming course from the Federal University of Campina Grande. One of the units of

the course approaches linear iterations over lists to apply tasks such as filtering and finding elements. In this context, consider a straightforward question asking students to find and return the index of the smallest element within a list. The expected design for this implementation is initializing the first element as the smallest and then iterating through the list, comparing each subsequent element to the current smallest. If a smaller element is found, the index of that element is stored. After checking all elements, the code returns the index of the smallest value in the list.

Once instructors typically take the well established approach of using Online Judges to assess students' code output, they are limited to checking if the output is the value expected, once they use black box tests (input/output). However, over the years, the instructors noted that some of the students were sorting the list and then returning the first element. Some of them implemented their own versions of classic sorting algorithms, while others—more concerning to the instructors—based their solutions on Python's built-in `sort()` function. In other words, while both Listing 2 and Listing 3 might pass the functional tests, they do not adhere to the specific design expectations set by the instructors.

Listing 2: Finding Smallest with Bubble Sort.

```
1 def get_smallest(sequence):
2   n = len(sequence)

3   for i in range(n-1):
4     for j in range(0, n-i-1):
5       if sequence[j] > sequence[j+1]:
6         swap(sequence, j, j+1)

7   return sequence[0]
```

Listing 3: Finding Smallest with Python's built-in *sort()* function.

```
1 def get_smallest(sequence):
2   sequence.sort()
3   return sequence[0]
```

The course instructors[1] believe that adhering to the expected design is crucial for the learning process, as it emphasizes key aspects like completing the task in linear time and avoiding side effects. Given the high volume of submissions, manually inspecting students' solutions is impractical. Therefore, there is a clear need for automated mechanisms, such as design tests, to check alignment between the expected design and the students' implementations.

[1]The last author was actively involved in the introductory programming course discussed in the paper.

Our Approach. To solve this problem, we have developed **PyDesignWizard**, a Code Structure Analyzer API that parses Python programs and offers methods to query detailed information about the analyzed code. This allows us to write design tests for scenarios like the aforementioned one. **PyDesignWizard** leverages Python's Abstract Syntax Tree (AST) for static analysis, integrating unit testing concepts with *Unittest* Python's built-in testing framework.

The **Python Design Wizard** API provides several methods that facilitate the automated inspection of algorithm structure, allowing instructors to verify whether students followed the expected design in their solutions. Some of the key methods include:

- **`get_calls_function(function_name).`** This method checks if a specific function, such as `sort()`, has been called within the student's code. It is useful for ensuring that students do not use built-in functions when the goal is to manually implement the algorithm.
- **`entity_has_child(parent_entity, child_type).`** This method checks whether a given entity (such as a loop or function) contains specific child elements, such as an assignment variable. It can be used to verify the complexity or modularity of the code.
- **`get_relations(relation_type).`** With this method, it is possible to inspect relationships between different code entities, such as nested loops or function calls. It allows the detection of complex patterns, like sorting algorithms that use multiple loops.
- **`get_entity(entity_name).`** This method identifies and returns a specific entity in the code, such as a function or class, enabling detailed analysis of that entity in terms of parameters, variables, and control structures.

Combined, these methods allow instructors to create tests that go beyond simple functional verification, enabling the evaluation of code structure and the detection of patterns or design violations expected in algorithm development.

Let us revisit the previous example and propose a design test to determine whether an implementation relies on a quadratic sorting strategy or Python's built-in `sort()` function. While one could write a test to check for specific patterns, such as the presence of a `for` loop, an `if` statement, or a variable storing the smallest element, the instructors in this case aimed to avoid overly restricting students' approaches. The key goal was to prevent the use of a sorting strategy

when a single iteration over the sequence would suffice. Therefore, the design test should focus on two main characteristics: the presence of nested loops, which may suggest a sorting algorithm, and the use of Python's `sort()` function. Listing 4 is the design test implementation for this using PyDesignWizard.

Listing 4: Design Test to check sorting strategies.

```
1 def test_sort_algorithm(self):
2   dw = PyDesignWizard("student-submission.py")

3   outer_loop = dw.get_entity('for1')
4   if outer_loop = None:
5     outer_loop = dw.get_entity('while1')
6   self.assertTrue(outer_loop != None)

7   inner_loop = outer_loop.get_relations('HASLOOP')
8   self.assertTrue(inner_loop != None)

9   change_list = dw.entity_has_child(inner_loop,\
      'assign_field')
10  self.assertTrue(change_list)

11  sort_calls = dw.get_calls_function('sort')
12  self.assertTrue(sort_calls == 0)
```

Line 2 creates an instance of PyDesignWizard, initializing it with the file "student-submission.py". When instantiated, the PyDesignWizard object analyzes the code's Abstract Syntax Tree (AST) to extract structural information, representing it as a set of entities and their relationships. It then offers methods for querying these entities and relationships, making it easier to examine the structure and behavior of the code under analysis.

Lines 3 to 6 attempt to locate an outer loop in the student's code, first checking for a 'for' loop (''for1'') and, if not found, checking for a 'while' loop (''while1''). The code then asserts that an outer loop was successfully found. If neither loop exists, the test will fail.

Lines 7 and 8 attempt to find an inner loop within the outer loop by checking for a ''HASLOOP'' relationship. If an inner loop is detected, the test continues; otherwise, the assertion in line 8 causes the test to fail.

Lines 9 and 10 verify whether the inner loop contains an assignment operation. Line 9 checks for a child entity labeled `assign_field` within the inner loop, indicating an assignment. Line 10 asserts that this assignment exists; if it does not, the test will fail.

Lines 11 and 12 check whether the `sort` function is called within the student's code. Line 11 retrieves the number of times the `sort` function is invoked using the method dw.get_calls_function('sort'). Line 12 asserts that the number of calls to the `sort` function is zero, indicating that the implementation should not rely on a built-in sorting function. If the assertion fails, it suggests that the student's code improperly uses the `sort` function.

Identifying the use of specific functions, such as the `sort()` function, is relatively straightforward and leaves little room for errors in the tests. However, note that the absence of nested loops does not necessarily mean that the student did not sort the sequence. They could have used different sorting strategies, such as Merge or Quick sort. Therefore, a design test may fail to detect some cases (false negatives), as well as flag instances where nested loops are used but are not actually sorting the sequence (false positives). For this reason, we evaluated our approach using these types of tests to assess precision and recall within the context of more complex scenarios.

4 EVALUATION

In this section, we evaluate our approach for writing tests that verify the program structure of students' code. We focus on three key aspects to assess its effectiveness and practicality.

First, we examine through unit tests our tool's ability to detect essential code elements. This involves analyzing how accurately the tool can identify specific components in students' code, such as loops, if statements, variables, commands, among others.

Second, we evaluate the performance of a specific design test in a more complex testing scenario that introduces uncertainty. This allows us to observe how well our approach handles ambiguous or incomplete information in the code, simulating real-world challenges where the structure might not always be clear.

Finally, we assess the feedback from instructors, particularly in terms of their ability to write tests using the proposed approach. We investigate whether they find the tool intuitive and useful, as well as their perceptions of how the approach can enhance the teaching and assessment process.

4.1 Methodology

The methodology of this study was guided by three key research questions:

- **RQ1.** Can the structure of students' solutions be effectively assessed through design tests?
- **RQ2.** Do educators perceive the concept of design tests as intuitive and easy to grasp?
- **RQ3.** Does a tool based on design tests offer tangible benefits in educational contexts?

In this section, we describe the two studies conducted to address the research questions. The first study focuses on the first question, while the second

addresses the latter two. These studies are as follows: (1) a quantitative evaluation of the API, assessing the confidence level in the test results produced by the PyDesignWizard, and (2) a qualitative study where professionals validate the tool's approach within the educational context.

4.1.1 Quantitative Study: Answering RQ1

Our API simplifies the identification of specific functions, data types, data structures, and other elements, with static analysis minimizing the potential for error. We have conducted extensive testing to ensure accuracy in these fundamental yet critical cases, which has instilled confidence in the reliability of our tool. To thoroughly assess the API, we generated synthetic Python programs that included various language constructs, such as loops, variable usage, data structures, and functions. We used the API to extract information from these programs and verified the correctness of the data retrieved. The unit tests we developed significantly mitigate risks associated with the tool's development. The results of our coverage tests can be summarized as follows:

- 33 unit tests;
- 100% function coverage; and
- Approximately 86% branch/code coverage.

However, we believe it is equally important to evaluate how design tests perform in more complex scenarios, where determining the implementation of a solution requires more detail and multiple approaches can be used to solve the same problem. To this end, we chose to assess the precision of a design test we wrote to detect whether a code implements a sorting strategy. We selected the sorting context for several reasons. First, sorting a sequence involves the use of loops, conditional statements, and data structure manipulation, making it a rich area for analysis. Second, there are numerous approaches to sorting, from classic algorithm implementations to leveraging pre-built functions. Lastly, over the years, instructors of the introductory programming course at UFCG have observed that students often resort to sorting algorithms to solve elementary problems where such complexity is unnecessary—a behavior the instructors seek to discourage. For these reasons, in this work we have evaluated the design test (Listing 4) detailed in the previous Section 2.

Dataset. We executed the design test in Listing 4 on a sample of 1,714 Python solutions, which we automatically collected from our Online Judge system. We focused on 21 specific problems, selecting these because they are the ones where students are most likely to use sorting as a shortcut to simplify the solution, rather than applying the appropriate strategy. Examples of these problems include: (i) finding the smallest or largest element in a list, (ii) checking if a list is already sorted, (iii) counting the number of distinct elements, and (iv) verifying if all elements are unique, among others.

Procedures and Metrics. After executing our selected Design Test, we manually analyzed the 1,714 solutions to ensure that the test accurately detected the presence of a sorting strategy. We then recorded the number of true positives, false positives, true negatives, and false negatives to assess the test's precision, recall, and accuracy. In this context, true positives indicate that the test correctly identified a sorting strategy in a solution where one was actually used. False positives occur when the test incorrectly flagged a non-sorting solution as employing a sorting strategy. True negatives reflect cases where the test correctly indicated the absence of a sorting strategy, while false negatives represent instances where the test failed to detect a sorting strategy that was actually present in the student's code.

Results and Discussion. Table 1 presents the confusion matrix used to classify test results in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).

Table 1: Confusion Matrix for Test Results.

	Detected Positive	**Detected Negative**
True Positive (TP)	49	9 (FN)
True Negative (TN)	15 (FP)	1695

The confusion matrix was used to calculate the precision, recall, and accuracy of the proposed approach. Precision, defined as the ratio of true positives (TP) to the sum of true positives and false positives (FP), was 77% (49 / (49 + 15)). Recall, which measures the ratio of true positives to the sum of true positives and false negatives (FN), was 84% (49 / (49 + 9)). Finally, accuracy, calculated as the ratio of all correctly classified instances (true positives and true negatives) to the total number of instances, reached 98,6% ((49 + 1641) / (49 + 15 + 1641 + 9)). These metrics demonstrate the effectiveness of the design test in correctly identifying sorting strategies while maintaining a low error rate.

The manual analysis of false positives revealed two main categories of programs that the test incorrectly identified cases as involving sorting:

- Operations that modify the iterated array: these operations involved changes to the array during iteration but were not related to reordering elements;
- Algorithms utilizing auxiliary structures: in these cases, the algorithm made use of additional data structures, such as dictionaries or arrays, which were initialized outside of the loop. These auxiliary structures were used for calculations unrelated to the array's ordering.

Listing 5 shows an example of false positive. Note that the code does have two loops, but it does not manipulate the list `numbers` to sort its elements.

Listing 5: False positive example.

```
def split_sequence(numbers, divs):
  for i in range(len(numbers)):
    num = numbers[i]
    for div in divs:
      if div != 0:
        num /= div
    numbers[i] = num
```

These findings highlight the challenge of precisely differentiating between true sorting strategies and other types of iterative modifications or calculations, emphasizing the need for further refinement in the design of the static analysis tests. Nevertheless, it is important to note that while the students were not employing a sorting strategy, they were still diverging from the design originally intended by the instructors, which did not require quadratic solutions for the given problem.

For the false negatives, the issues seem to be largely related to the absence of test checks for certain scenarios. In 8 of the false negative cases, the students used either native functions or custom implementations that performed operations on the list, which the test failed to detect. Examples include the use of functions like `insert`, `pop`, or custom implementations of such operations. In one particular case, a student utilized a lambda function, and our API did not capture this instance due to the lack of implementation for handling the structure of lambda expressions.

Listing 6 shows an example of false negative. Note that the student implemented the insertion sort, but used the insert function instead of moving the elements applying swaps (`assign_field`).

Listing 6: False negative example.

```
def my_insertion_sort(arr):
  for i in range(1, len(arr)):
    value = arr.pop(i)
    j = i -1
    while j >= 0 and arr[j] > value:
        j -= 1
    arr.insert(j+1, value)
  return arr
```

The limitation encountered regarding the false negatives was not inherent to the API itself, but rather in how the test was written. The test could have been more specific, explicitly including functions like 'insert' and 'pop', which were overlooked. However, this raises an important discussion about the balance between writing highly specific tests that aim to capture many details, and more general tests that, while simpler, are prone to false positives and false negatives. Finding the right balance depends on the goals of the test and the context in which it is applied.

On the other hand, the case involving the lambda function highlights a genuine need to evolve the API. This example indicates that the current version of the API does not fully capture certain language features, and extending it to handle lambda expressions would improve its overall accuracy and coverage.

4.1.2 Qualitative Study: Answering RQ2 and RQ3

Relying solely on professionals' impressions of the tool's documentation would not provide sufficient confidence to address our research questions concerning the usefulness and usability of the solution. For this reason, we conducted a qualitative study in which we interviewed 16 professionals in order to evaluate and collect deeper insights on the design test approach for education. Next, we give details on the participants, the procedures applied to collect and analyze data, and the results of this study.

Participants. We have interviewed 16 instructors from three different universities of Brazil. The participants were invited through mailing lists sent to three different universities in Brazil, ensuring a diverse representation of institutions. They are all professionals in the field of education, with direct experience teaching programming-related courses. This group includes both university professors and graduate students who serve as instructors for programming disciplines. From now on, we will refer to these participants as P1, P2...P16.

Interviews and Data Collection. In addition to testing our API, we also evaluated whether instructors are capable of understanding the concept of design tests, writing their own design tests, and their perspectives on using this approach in an educational context. To this end, we asked instructors to perform two tasks: (i) reading 3 pre-written design tests and

explaining these tests, and (ii) writing 4 design tests based on specific scenarios we provided.

The 3 pre-written tests were chosen for their increasing complexity, starting with a test for simple function calls and progressing to more complex structures such as loops and sorting algorithms, as sorting algorithms are central to verifying structural understanding of code. This choice was strategic, as the third test, which required detecting sorting algorithms, allowed us to see whether instructors could understand and explain a more advanced concept. In the task of writing 4 new tests, the scenarios provided ranged from verifying variable assignments to checking for correct usage of functions and loops. These tasks were selected to reflect common educational situations, ensuring the instructors could apply the concept of design tests in practical settings.

During the evaluation, we employed the think-aloud protocol (Ericsson and Simon, 1993b) to collect data on their experiences while performing the tasks. The think-aloud protocol is a qualitative research method where participants verbalize their thought processes in real-time as they complete a task. This approach helps capture the reasoning, difficulties, and strategies used by participants, providing insight into how they interact with the material. Each session lasted approximately 20 minutes on average. We recorded the sessions, including both the task performance and follow-up interviews, and later transcribed the audio. These transcriptions were reviewed for accuracy and then subjected to coding. We worked with predefined tags related to the usability of the API and the understanding of design tests to ensure consistency in categorizing the responses. Coding is a systematic process used in qualitative research to categorize textual data into themes or patterns. By applying coding, we were able to organize participants' responses into meaningful categories that facilitated our analysis of how well the instructors understood the design testing process, how they approached writing tests, and their overall impressions of the approach.

Results. We matched participants' verbalization to 7 pre-defined tags: Positive feedback (tool or approach), Negative feedback (tool or approach), Difficulty (use or understanding), Ease (use or understanding), Correct response, Incorrect response, and Response close to correct. The observations from the data are the following.

Observation 1: Instructors Can Easily Understand the Concept of Design Tests.

The majority of the instructors quickly grasped the concept of design tests after a brief introduction. Out of 16 participants, 15 reported understanding the concept easily, and 10 instructors were able to think about implementations even before consulting the documentation. This highlights the intuitive nature of the concept for professionals in education.

The qualitative data collected shows a significant amount of positive feedback. There were 103 comments tagged under "Ease of Understanding". 8 instructors asked questions or made comments that showed a deeper engagement with the idea of design tests, suggesting a high level of comprehension. On the other hand, one participant (P12) could not easily understand the approach and described it as confusing.

P1: *"From what I understand, 'for' is also an entity, so in this line here, I am grabbing all occurrences of 'for' and returning a list with those occurrences, and I check if they are different from an empty list."* This demonstrates the participant's ability to interpret the role of programming structures as entities in the design tests and understand their functional use within the tool.

P2: *"[...] I really liked it, I found the way of doing these tests interesting, I think it's cool. My feedback is entirely positive."* This feedback emphasizes the enthusiasm for the design test approach and its perceived usefulness.

Observation 2: Design Tests Are Seen as a Valuable Tool for Educational Use.

Several instructors mentioned that they could see immediate applications for design tests in their teaching. Out of the 16 participants, 8 directly pointed out potential educational uses, such as automating the structural correction of student assignments. Furthermore, 4 instructors stated they would consider integrating the tool into their teaching routines, particularly in disciplines involving programming practices.

P3: *"Thinking quickly, for correction, the first thing that comes to mind is grading exams."* This shows that instructors quickly identified practical applications for design tests, specifically for automating and enhancing the feedback process on student submissions.

P4 reflected on the broader use of the tool in programming courses: *"I think in the Programming II course, it could be interesting to use these design tests along with unit tests, because they abstract the idea of what needs to be tested."* This indicates that design tests could be a valuable addition to traditional unit tests, providing a complementary layer of feedback that focuses on the structure of students' code.

Observation 3: Some Instructors Experienced Minor Difficulties with the API, but the Overall Concept Remained Clear.

While the majority of participants found the concept

of design tests easy to understand, a few noted minor challenges when working with specific API functions. For instance, 6 instructors expressed some confusion about how to manipulate certain relationships between entities, which slowed down their ability to implement the tests.

P5: *"I had some doubts about how to manipulate some of the relationships between the entities, but I think it's something that can be understood with more practice."* Despite these difficulties, the participants generally found that the concept was clear and the tool was accessible.

P6: *"I didn't understand the last one, do you want me to check if there's at least one 'for' inside 'func1'?"* This indicates that while there were occasional misunderstandings, they were more about the technical details of implementation rather than the core concept of design tests.

Observation 4: Instructors See Potential in Combining Design Tests with Other Educational Tools. Some participants reflected on how design tests could complement existing tools and methodologies in education. For example, instructors suggested combining design tests with unit tests to provide more comprehensive feedback to students. This would allow educators to not only check the correctness of a solution but also assess the structure of the code.

P7: *"I think design tests could be used to detect when students are using functions they shouldn't, like built-in libraries."* This suggests that design tests can help enforce specific coding guidelines and encourage students to focus on learning foundational programming concepts.

In conclusion, the feedback from instructors strongly supports the observation that design tests are an intuitive and valuable tool for educational contexts. The concept was widely understood, and instructors were able to think critically about its application in programming education. Despite minor challenges with the API, the overall reception was positive, with participants expressing interest in incorporating design tests into their teaching routines.

5 RELATED WORK

The automatic assessment of programs in introductory programming courses has been extensively studied (Ala-Mutka, 2005) (Paiva et al., 2022) (Keuning et al., 2018) (Wasik et al., 2018). One of the most common approaches is the use of Online Judges, which automate the verification of functional correctness through input-output tests. Specifically, Janzen et al.(Janzen and others, 2013) and Singh et al.(Singh et al., 2013) propose the use of automated judges to verify whether students' programs produce the expected results for a given set of test cases. However, this method is limited to verifying the program's output, without considering internal aspects such as the use of control structures, recursion, and modularity.

On the other hand, Truong et al. (Truong et al., 2004) explore the application of static code analysis in an educational context, investigating how verifying characteristics like cyclomatic complexity, code coverage, and programming best practices can contribute to automated assessment. However, this approach focuses on code quality in general, without specifically covering algorithmic structure.

Araújo et al. (Araujo et al., 2016) emphasize the need for more robust methods that consider not only code quality but also the algorithmic structure of programs. They demonstrate that, in the absence of clear guidelines, many students adopt solutions that, while functionally correct, do not adhere to the design criteria expected by instructors, hindering the development of logical and abstract reasoning skills.

This work aims to address the limitations of these existing approaches by proposing a design verification tool that uses static analysis to detect structural deviations in students' solutions. Thus, the tool complements functionality-based assessments, enabling educators to provide more comprehensive feedback.

6 CONCLUSION AND FUTURE WORK

In this work, we introduced the **Python Design Wizard**, a tool designed to inspect the structural aspects of algorithms in programming courses. Our approach focused on the use of **design tests** to verify not only the functional correctness of students' code but also its structural design, ensuring that students adhere to the expected algorithmic patterns. Throughout the study, we evaluated the effectiveness of the tool in identifying violations of algorithm design, with promising results.

Our tests, particularly in detecting **sorting strategies**, demonstrated a high accuracy of 98.6%, with a precision of 77% and recall of 84%. These results reinforce the potential of design tests to supplement traditional unit tests by addressing aspects related to the internal structure of algorithms, such as control flow, recursion, and the use of specific functions.

6.1 Threats to Validity

While the results were promising, there are certain threats to the validity of our approach. First, the sample size was limited to 1714 student programs from a single institution, which may not fully represent the diversity of approaches found in broader educational contexts. Additionally, the focus on sorting algorithms may have limited the generalizability of the tool's applicability to other algorithmic topics.

Moreover, the qualitative evaluation was conducted with a relatively small group of instructors, which may not fully reflect the perspectives of a wider range of educators. Future studies should consider expanding the qualitative evaluation to a broader sample of instructors and different educational settings.

6.2 Future Work

As future work, we plan to expand the functionality of the Python Design Wizard by creating a catalog of pre-defined design tests that can be shared with the community of educators. These tests could be tailored to various common topics in introductory programming courses, such as recursion, dynamic programming, and data structures.

Furthermore, it would be interesting to investigate the integration of this tool into other programming paradigms and languages, as well as expanding its capabilities to more complex scenarios that go beyond sorting algorithms. We also aim to explore the possibility of developing more specific tests that could handle different levels of complexity, providing a more granular evaluation of algorithm design.

An interesting observation from our study is that we evaluated the tests themselves rather than the tool directly, which is why we included unit tests in our validation process. This approach allowed us to measure how well the design tests perform in identifying algorithmic patterns and violations, rather than focusing solely on the technical capabilities of the tool.

REFERENCES

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102.

Araujo, E., Serey, D., and Figueiredo, J. (2016). Qualitative aspects of students' programs: Can we make them measurable? In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE.

Beazley, D. (2012). *Python Essential Reference*. Addison-Wesley.

Brunet, J., Guerrero, D., and Figueiredo, J. (2009). Design tests: An approach to programmatically check your code against design rules. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 255–258.

Ericsson, K. A. and Simon, H. A. (1993a). *Protocol Analysis: Verbal Reports as Data*. MIT Press, Cambridge, MA, revised edition edition.

Ericsson, K. A. and Simon, H. A. (1993b). *Protocol Analysis: Verbal Reports as Data*. MIT Press, Cambridge, MA, revised edition edition.

Foundation, P. S. (2023). *Python 3 Documentation*. Available at https://docs.python.org/3/.

Janzen, S. and others (2013). Automated grading systems in competitive programming contests. In *Proceedings of the 8th International Symposium on Educational Software Development*.

Juiz, A. and others (2014). Online judge and its application to introductory programming courses. In *ACM SIGCSE Bulletin*.

JUnit Team (2023). *JUnit 5: The Next Generation of JUnit*. Accessed: 2024-11-11.

Keuning, H., Jeuring, J., and Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43.

Leite, L. (2015). A proposal for improving programming education with frequent problem solving exercises. In *Brazilian Symposium on Computer Education*.

Paiva, J. C., Leal, J. P., and Figueira, Á. (2022). Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, 22(3):1–40.

Papastergiou, M. (2009). Digital game-based learning in computer programming: Impact on educational effectiveness and student motivation. *Computers & Education*.

Singh, J., Gupta, P., and Sharma, M. (2013). Online judge system for learning and practicing programming. In *International Journal of Computer Applications*.

Truong, N., Roe, P., and Bancroft, P. (2004). Static analysis of students' java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 317–325. Citeseer.

van Rossum, G. (2009). *The Python Language Reference Manual*. Network Theory Ltd.

Wasik, S., Antczak, M., Badura, J., Laskowski, A., and Sternal, T. (2018). A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, 51(1):1–34.

Watson, C. and Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44.