Interactive ATM Interface Using Java

AJAY KUMAR[1], ABHINAV VERMA[2], SANDEEP PANDEY[3], SAMEER MISHRA[4]
[1]*Faculty, Inderprastha Engineering College, Ghaziabad*
[2, 3, 4]*Student, Inderprastha Engineering College, Ghaziabad*

Abstract*— Increasing the effectiveness of programming education has emerged as an important goal in teaching programming languages in the last decade. Automatic evaluation of the correctness of the student's source code saves teachers time and effort and allows a more comprehensive focus on the preparation of assignments with integrated feedback. The study aims to present an approach that will enable effective testing of students' source codes within object-oriented programming courses while minimizing the demands on teachers when preparing the assignment. This approach also supports variability in testing and preventing student cheating. Based on the principles of different types of testing (black-box, white-box, grey-box), an integrated solution for source code verification was designed and verified. The basic idea is to use a reference class, which is assumed to be part of every assignment, as the correct solution. This reference class is compared to the student solution using the grey-box method. Due to their identical interface (defined by assignment), comparing instance states and method outputs is a matter of basic programming language mechanisms. A significant advantage is that a random generation of test cases can be used in such a case, while the rules for their generation can be determined using simple formulas. The proposed procedure was implemented and gradually improved over 4 years on groups of bachelor students of applied informatics with a high level of acceptance.

***Indexed Terms*- Automatic testing, electronic learning, engineering education, software testing.**

I. INTRODUCTION

Achieving greater efficiency in programming teaching has become one of the essential goals in the field of programming language teaching in the last decade. The primary reason is the growing need for programmers in the labour market and the increasing demand for many programming languages. The programmer must learn to think in a specific programming language and main them. Furthermore, the extension of MOOCs and e-learning courses has created the preconditions for the implementation of assignments that allow the automatic evaluation of the correctness of source codes by electronic tools and can be used in large groups of students.

Currently, many researchers are exploring the potential for AI models like ChatGPT to take over the role of programmers [1], [2]. This idea is often elaborated in unscientific sources, often leading to exaggerated claims. However, the reality is that the displacement of programmers by AI is primarily relevant to tasks that involve automated procedures and the creation of basic code snippets [3]. AI models hold significant potential as helpful tools for programmers when dealing with clearly describable and frequently repetitive code segments. However, it's currently challenging to envision them being able to design a complex, efficient, and reliable application.

Therefore, the objectives of training programmers remain consistent compared to the previous decade. Computer pro- gramming is important in gaining problem-solving skills and critical thinking development [4]. Acquiring programming skills enhances students' ability to express themselves clearly and solve problems accurately, extending to areas outside of programming [5]. Integrating the complementary capabilities provided by generative AI into programming education is desirable at a certain stage of preparation.

Research-based recommendations usually state that teach- ers should be available to students while they are solving the automatically evaluated tasks. The teacher's role is to help students and explain the feedback of the wrong result or perhaps the whole problem that the task should solve. The reality, however, is that these types of systems are primarily used in self-study [6] or flipped classrooms [7], [8].

Based on [9], important reasons to prepare educational environments that support learning programming are:

- To make progress, students need help because

learn- ing programming is challenging, especially in the introductory phases of new topics. The environments provide immediate feedback to students and support their learning speed.
- Programming courses are attended by thousands of students worldwide and by hundreds of students at individual universities. Helping students individually and solving their repetitive problems requires enormous time and teachers' commitment. The aim of educa- tional environments is a reduction of the teacher's workload focused on evaluating assignments and the ability to move the saved time to another area of his activity.

The time and effort of the teacher saved by the automatic check of the correctness of the student solutions should be devoted to the thorough preparation of assignments integrating feedback, especially in case of incorrect answers. The paper aims to describe a universal structure and form for verifying the correctness of the source code so that the student receives clear and precise information about which part of the program is incorrect and why.

The article consists of several parts. The theoretical background aims to provide a comprehensive overview of current testing methodologies. It starts with a brief description of static testing and then moves into dynamic code analysis. The focus then shifts to thoroughly examining black, white, and grey-box testing principles. This survey covers various approaches, techniques, advantages, and dis- advantages to identify appropriate methodologies applicable to the presented research. In addition, this chapter covers the process of generating test cases.

Related work is devoted to generating feedback and evaluating the code in the virtual educational environment.

The core of the paper is focused on the conception and validation of an approach adapted to facilitate effective testing of student source codes in object-oriented programming.

The final section summarizes the primary findings, con- textualizes them, and discusses potential applications of the presented approach.

II. THEORETICAL BACKGROUND

Evaluating the correctness of source code brings several challenges. Some of them have already been solved, and some of them have partial solutions. Source code correctness testing can be implemented at two levels [10], [11]:
- Static analysis is based on source code analysis. It focuses purely on the structure of the program and returns a measure of consistency with the required specifications. It does not automatically follow that a program that returns the expected results is also statically correct – it may use the wrong program structure or some inefficient items.
- Dynamic analysis involves executing code that itera- tively runs for different input parameters. The param- eters are to be chosen to cover both critical and general values, and the correctness of the program is determined by matching the results obtained by executing the program with the predefined correct values.

A. *STATIC ANALYSIS*

The information from the static analysis is used to improve code quality, safety, and robustness and, in some cases, to identify cyclic method calls or infinite code loops. In addition, according to [12], static analysis aims to identify syntax and interface errors, reveal the potential for code reduction, highlight architectural standards, remove potential sources of bugs and inefficiencies, and eliminate actual bugs. According to [13], typical examples of errors and problems uncovered by static analysis are uninitialized variables, unreachable code, unused variables, type mismatches, and many other mistakes often arising from programmer in attention. Therefore, the authors collected and processed information on applying static analysis to code written by beginners and then quantified and analysed the identified errors.

The results confirm that novice programmers make many errors such as ‘‘uninitialized variable’’, ‘‘unused variable’’, ‘‘type mismatch’’, and ‘‘index out of list scope’’. Most of these errors go unnoticed by novices and are often the cause of incorrect solutions, even if the original idea of the algorithm is correct. The study shows that integrating static analysis into the learning process can make a difference, especially since it can help novices find many common mistakes they make before running the code. Furthermore, the mentioned functionalities were often implemented as features in modern development environments as hints for programmers.

However, identifying whether the code is correct or incorrect cannot remain on static analysis. According to [14], current static analysis tools can provide developers with enough information to assess what to do with the warnings generated, but they rarely offer a relevant fix for what they claim is a problem. Static analysis tools offer quick fixes, provide a potential solution, and apply it to the problem to help developers assess warnings more quickly and ultimately save time and effort.

B. *DYNAMIC ANALYSIS*

Dynamic code analysis has its place in programming educa- tion and real-world application development. The principle of data preparation and testing is very similar in both cases. The two basic approaches to code testing are built on knowing or not knowing the internal structure of the unit (program or class) under test:

- White-box testing - enables in-depth analysis and bug detection in the internal program structure, assisting users in optimizing the program to its full potential [15].
- Black-box testing - the code being tested cannot be accessed (does not need to be accessed) by the tester; this approach checks the dynamic behaviour of the program and is generally faster and easier to perform than white- box testing.
- Grey-box testing - combines the advantages of both testing methods and creates a kind of intermediate layer between the accessibility and the unavailability of the tested code. In this case, the tester has a description of the interface of the tested code but does not know its exact implementation.

1) BLACK-BOX TESTING

Several perspectives and parameters define black-box and white-box testing categories. According to [16] and [17], the black-box approach assumes that the program is considered a ‘‘big black box’’ for the tester, who cannot see inside the box. All a software tester knows is that inputs can be given to a black box, and the black box processes them and sends ‘‘something’’ back. The primary objective is to analyse the generated output and evaluate its compliance with the desired specified requirements and/or outputs.

This communication usually uses an I/O (input/output) approach to verify the correctness of simple programs. The program reads the values appearing as inputs from the user and returns the corresponding values as output to the console. Both inputs and outputs are redirected so input and output reading can be automated. The validating software then compares whether the expected results follow the outputs obtained based on the execution of the program (Figure 1).

The most crucial step in black-box testing is the proposal of suitable test cases, emphasising covering all types of program behaviour and potential exceptions. Techniques aimed at streamlining the test preparation process and maximizing input coverage can be categorized into several groups within the context of testing:

```
case = Test6
input = 598 37490
output = 13/815
case = Test7
input = 174 5202
output = 29/867
```

FIGURE 1. Example of data for I/O testing (a truncation of a fraction).

- Boundary Values Analysis is based on the finding that many errors tend to occur near extreme values of the input variables. For each input variable, it is therefore desirable to carry out tests containing the minimum value, values just above the minimum value, ‘‘normal’’ values, values just below the maximum value and the maximum value [18]. Sometimes, it is necessary to verify the correctness of the program’s behaviour with values below the minimum and above the maximum value. Test cases are designed for valid and invalid boundary values [19].
- Fuzzy Testing is a testing technique that focuses on creating random inputs and monitors their execution to identify errors and unusual program

behaviour [20]. The main goal is to detect errors, unverified inputs and vulnerabilities by making simple, invalid or unexpected inputs. The fuzzy testing process is automated and generates inputs randomly or based on specified patterns [21], [22].

- Equivalence Partitioning is a technique that aims to reduce the number of test cases, dividing inputs into groups so that each focuses on a specific property or part of the program (e.g., branches in the code, boundary values, etc.) [23]. The technique is based in part on the assumption that all inputs in the group cause similar behaviour (and errors) in the program. That is, if one input value from the group causes a failure, then it is assumed that the remaining inputs from it are also problematic, and conversely, for inputs that do not cause an error, the entire group can be considered successfully tested [24].
- Pair-wise Testing (All Pair Testing) is based on the idea that most software errors show up in specific combina- tions of input values, and all possible combinations of inputs can be reduced by identifying the relationships between them. This type of testing is based on the idea that merging two or more testing parameters could reduce test cases, and a software system can be tested more proficiently and quickly while covering all real test cases. Pair-wise testing has limitations in situations where errors depend on combining three or more input parameters. In that case, other techniques must be used (e.g. random selection of inputs) [25].
- Cause-Effect Graphis a testing method with a heuristic approach [26]. It is the only black-box testing technique that considers combinations of causes for system Therefore, if the previous test cases did not reveal the failure, the new test cases should be far away from the already executed test cases that do not cause the failure [61].
- Search-based testing is the combination of automatic test case generation and search techniques. The search algorithms aim to automate the process of prioritising test cases, generating test data, optimising software test oracles, minimising test suites, authorising real- time properties, etc. [66]. A problem-specific fitness function that guides searching for good solutions from a potentially infinite search space in a practical time limit is crucial to optimisation. A typical example is using genetic algorithms to prepare test cases [67].
- Metamorphic testing is a technique that uses some necessary properties of the software under test to create new test cases [63]. New test cases are based on transforming some selected existing test cases. For the test result verification, instead of using an oracle for each test case, the test results from multiple test cases are checked against the corresponding metamorphic relations [68].

Each technique has advantages, and limitations must be carefully considered per project-specific requirements. The effectiveness of the testing process and its ability to detect potential flaws depends on choosing an appropriate test generation approach. Combining these techniques can provide better coverage of test scenarios and improve overall software quality.

III. RELATED WORK

Integrating software testing into the software engineering curriculum is essential [69]. In addition, when teaching the basics of programming, it serves as a key tool that allows students to receive accurate feedback on the correctness and accuracy of their code. Often used by educators and course creators, this approach makes it easier for students to understand programming principles better and improve their skills.

In practical implementation, this methodology works in such a way that students develop their programs according to the assignment. The programs are then verified using a testing tool. Tests are prepared in advance; evaluation usually involves sending the program and providing feedback [70]. The results of the evaluations provide students with an insight into the correctness of their code and its alignment with the required functionality.

This approach has several advantages. First, it lets students see how programming theory translates into program code. It also offers instant feedback, allowing students to quickly correct mistakes and improve their programming skills. Finally, it teaches students the importance of code quality control and error detection principles.

Overall, using testing in learning the basics of program- ming helps students better understand the practical aspects of programming and increases the quality of their programs.

Thinking about source code evaluation in learning pro-gramming makes sense if tests can be automated, quickly executed, and run anytime and many times. All the presented approaches fulfil the fundamental prerequisites for integrat- ing into students' instructional curriculum. Nevertheless, it is necessary to consider the natural flaws that could potentially weaken their effectiveness:

- It is not easy to design test cases if the student assignments are not clear and concise.
- If the tester is not paying attention, it is possible to create redundant tests, which can give the impression of code correctness [19].
- Black-box (I/O) testing is challenging to define for more composite and complex tasks.
- Black-box (I/O) testing does not apply to testing code segments without changing the evaluated code.
- It is very expensive (time, code) to prepare white-box tests [71], [72].
- Although most platforms have tools for creating and executing tests, they are not available for every kind of implementation/platform [19].
- The test code must also be modified if the verified code has been changed (addition, change of parameters/ logic).

The shortcomings of the I/O approach are partially eliminated by unit testing, but the limitations resulting from the natural effort of students to simplify the solving of hard tasks should be solved with a more advanced technique.

An example of a student solution exploiting validation through I/O access is in Figure 2. Even though the code for the correct solution is much shorter, the student chose the cheating route.

```
n = str(input())
if n == 'BEWARE of bites':
 print('EAEo ie')
else:
  if n == 'MONKEYS are here':
    print('OKY r ee')
  else:
    if n == 'BANANAS are hanging on a tree':
      print('AAA r agn nate')
```

FIGURE 2. A typical student fraud attempt for the task ''Write characters in odd positions.''

A. *FEEDBACK GENERATION*

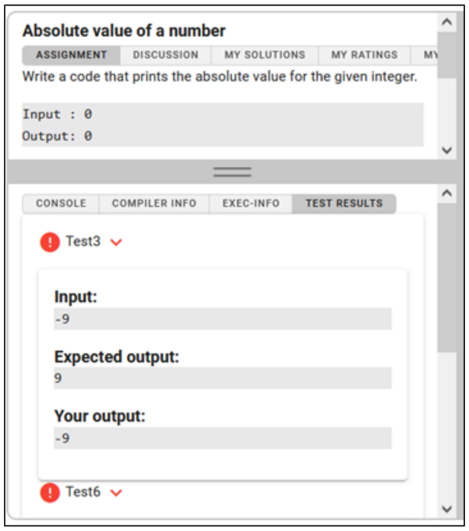
Getting feedback is an essential part of any learning process. Based on [9], feedback can be considered a formative method supported by acquiring knowledge and skills. Valuable feedback can provide the learner with corrective information, provide alternative solutions, bring information to clarify ideas, provide encouragement, or confirm that their answer is correct [73]. Quality feedback enables an overall increase in the quality of educational content. The types of feedback can be looked at from different angles.

A brief view of the main characteristics brings [74] looking at the feedback from the following aspects: groups of users in an electronic course or a standard study program is exceptionally difficult and inefficient in terms of effort and effect.

B. *EVALUATION IN THE EDUCATIONAL ENVIRONMENT*

Even though research pushes the boundaries of source code evaluation and partly automates feedback generation, reality lags behind the most modern ideas. The content creator (often a university teacher) must work with time- efficient tools, can be handled by programmers when creating assignments, and support the creation of new content even with evaluation mechanisms in a short time. Dozens of educational systems or plugins are currently available [9]. They support verifying source codes using some forms in a defined set of programming languages.

Dozens of systems with (automatic) program task eval- uation use an input-output (black-box) approach. The correctness of the solution is evaluated by checking the student program outputs with prepared input values and expected outputs, which are manually inserted into the evaluation system by the assignment creator. This approach is popular; it allows different levels of penalties according to the ‘‘importance’’ of specific inputs in many systems, but it usually only evaluates the program as a whole, not its parts [87]. Figure 4 provides a typical view of a partially correct program.


Absolute value of a number

Write a code that prints the absolute value for the given integer.

```
Input : 0
Output: 0
```

Test3

Input:
-9

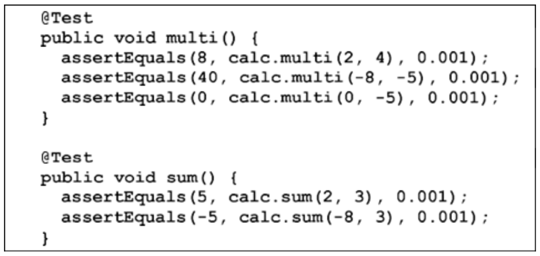
Expected output:
9

Your output:
-9

Test6

FIGURE 4. Example of evaluation result (Priscilla system [45]).

An alternative, again frequently used solution, is integrat- ing unit testing into education systems. The xUnit libraries (e.g., JUnit, etc.) are used to check the correctness of the code using a white-box testing approach. They allow the definition of the test cases and expected correct results at the level of the entire program and the level of individual class methods. The advantage of this approach is the ability to define the accuracy of the result, which is especially useful if the solution results are real numerical values and the ability to test program parts. However, the result is again compared against specific expected output values. Figure 5 presents the verification of methods for multiplying and adding a pair of numbers with the accuracy of 0.001 required when working with real numbers.



```
@Test
public void multi() {
  assertEquals(8, calc.multi(2, 4), 0.001);
  assertEquals(40, calc.multi(-8, -5), 0.001);
  assertEquals(0, calc.multi(0, -5), 0.001);
}

@Test
public void sum() {
  assertEquals(5, calc.sum(2, 3), 0.001);
  assertEquals(-5, calc.sum(-8, 3), 0.001);
}
```

FIGURE 5. JUnit testing – tests definition.

Testing frameworks are robust and proven tools that support the automation of the testing process and accelerate the deployment of modified code in production. Code testing for borderline, risky and general inputs is generally sufficient, and it is possible to use the same input values all the time.

In all approaches, students receive feedback consisting of input values, student program output, and expected correct output values. However, repeated use of the same inputs without additional teacher control or additional mechanisms focused on source code analysis may lead students to cheat and confidently generate input-based results using a simple condition (Figure 2).

However, in the case of using the principles of xUnit frameworks in verifying the correctness of student assign- ments, it is appropriate to make certain adjustments aimed at simplifying the preparation of projects, preventing cheating, and providing clear and understandable feedback.

Although standard elements of grey-box testing can be applied in educational environments, incorporating tools that aid in test creation can indeed be complex. Various annotation languages, such as JML for Java, Spec# for C#, and ACSL for C, can significantly assist in formulating and executing tests and defining other essential properties within the system. At the validation level, these languages can also contribute to generating black-box test data, ensuring comprehensive testing coverage [88]. However, integrating these tools into educational systems can present challenges due to the technical integration process and the specific requirements of educational environments.

Educational systems often work in web environments and communicate with intricate evaluation systems. Test creators must ensure the swift execution of tests to accommodate the simultaneous usage of the system by numerous users, thereby minimizing processing time and data transfer volume. This problem introduces additional performance, scalability, and efficient resource utilization challenges.

IV. CASE STUDY

The following case study shows the potential of implement- ing a robust source code verification mechanism within a methods. This difference was identified because grey-box methods require the program to be successfully compiled and run to facilitate evaluation, while white-box strategies allow evaluation using static evaluation methods. The result of the study confirms that context can have benefits beyond simply solving a given problem correctly.

Garcia-Magarino et al. [93] present a novel online judge system designed to evaluate long programming practices based on unit testing for small pieces of code (e.g. functions, methods, and classes) and provide specific comments for each failure to guide students. Climent & Arbelaez [94] present a case study focused on teaching OOP principles, using Python as a programming language and unit testing to verify code correctness.

Fischer and Von Gudenberg [95] use their code to compare the student and tutor class method results. This approach dramatically simplifies the test-writing process, as it enables the utilization of standard object manipulations without necessitating explicit and intricate checks.

Staubitz et al. [96] point out that in the case of massively parallel usage and code verification, response times quickly reach a level where it is no longer possible to provide timely feedback to students. Based on the knowledge and methodologies used in the available solutions, unit testing stands out as the most practical approach to implementing grey-box testing. This approach is feasible for most frequently used programming languages through xUnit test components based on industrial software testing principles [97]. A typical process of execut- ing the xUnit-based test is presented in Figure 6.

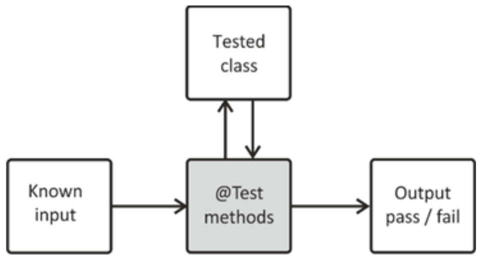


FIGURE 6. JUnit testing process [98].

To verify the class correctness, it is necessary to verify its behaviour in the following situations:

- The functionality of nodes for inputs – constructor, setters with typical, boundary, and prohibited values.
- The functionality of output nodes – getters for identifying the content of attributes or structured output of class state with appropriate output formatting.
- The functionality of methods – method outputs if
- it returns output, attribute state changes due to the application of methods that do not return results.
- The functionality of sequence of methods – verification of the correctness of the logic of the class, whether the sequence of methods correctly affects or, on the contrary, blocks changes made to attributes.

Tested methods within testing are isolated by default, and created instances are not affected by this. As part of preparing test frameworks based on xUnit focused on education, several tools supporting primary educational aspects were created [80], [99]. The test creator defines a list of values (parameters) for the input to the tested method and adds the expected result values to it (Figure 5). The precision can also be determined if a difference between the expected and the return value is assumed.

Running a test consists of the sequential execution of methods comparing the expected result and the result obtained from the instance of the tested class. The test framework ensures the independence of the tests from each other. According to the configuration, it creates a new instance for each test or performs other defined behaviour.

The result of the testing process is a list of tests with information on which tests were successful and which were not.

However, in the case of using the standard *assertEquals* methods, the following problems are encountered: • Writing tests and finding the correct outputs for manually entered inputs is laborious; moreover, in some cases, to obtain the output, it is necessary to combine the method performing the given operation and the getter.

- Tests use the same values repetitively, which can encourage students to cheat.
- Standard methods built into xUnit library test routines do not provide feedback in the form of recommendations. To solve the mentioned problems, the approach of greybox testing will be used, and random input generation and orthogonal array testing with certain rules will be applied.

D. *PROPOSED TESTING APPROACH*

The proposal is based on requirements aimed at the necessity of using different variable values and informing the student about the reason and location of the error. Due to the necessity of comprehensive verification of the methods in the tested classes, it is necessary to start from the principles of white or grey-box testing. The elimination of the shortcomings resulting from the use of the white-box approach is based on the following facts:

- Because code testing is a part of the learning process, it is desirable to have a correct (reference) solution created for each assignment. The reference (and correct) solution can be tested with the same set of tests as the student solution. To verify the correctness of the student's solution, it is, therefore, the best approach to compare the results of the methods of the student's class with the results of the teacher's solution.
- Preventing cheating or making testing more interest ing can be achieved by generating random values or selecting a random value from a predefined list. All tests should be monitored and tracked to gain inputs causing unexpected behaviour (following the requirements of identifying and logging problematic cases [100]). Identified problems can be solved by limiting input conditions or reformulating the statement for further use.
- Each test can inform students what it tests through its name. Notes, recommendations, or an explanation of the reasons for the wrong result of the test can enrich the essential information as an additional object or text.

The following input requirements are necessary for a successful implementation of validation:

- detail and unambiguous assignments
- reference class (correct solution)
- test cases focused on:
- constructors
- complex output(s) of class attributes
- setters
- getters
- methods
- the predefined or randomly generated sequence of
- methods simulating the work of a class in a real environment
- code ensuring the comparison of the reference and student class results.

When creating test cases, it is necessary to incorporate the utilization of boundary value analysis, equivalent partitioning techniques, and orthogonal array testing. These techniques cannot be automated in the simplest form of a test tool, but the form of random input generation presented below supports them.

The structure presented in Figure 7 was designed and verified during several cycles of educational courses to cover the entire process.

The inputs to the evaluation are the group of classes representing the student's solution and the group of classes representing the teacher's (reference) solution. The solution of the assignment can be one class, a group of independent classes that communicate with each other, classes connected by inheritance or another relationship. For simplicity, the student and the reference solutions will be referred to only as individual classes. The last part of the input is either a configuration file or a class that contains constants with detailed instructions for generating parameters in individual test cases.

Following the description of the testing process for JUnit presented in [101], the complete test run has two phases.

The first phase is the configuration and preparation of test values based on the test case description. Input values can be of different types, adapted to collections that code information for their random generation during testing. Based on the list of test cases, the creation of tested instances in the Main class is then prepared, where the whole process is started. Finally, the provided test cases are verified in independent tests.

The second phase is test case execution. This phase consists of the following:

1) Start of the process - the *Main*class instantiates the *Test Evaluator* class. Depending on requirements, a separate

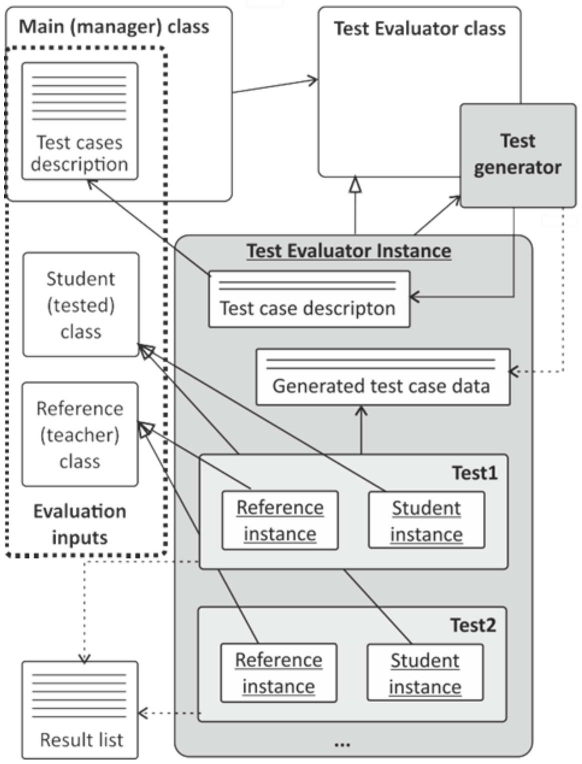


FIGURE 7. Simplified scheme of test generation and collection of test results.

instance can be created for each test (following the requirement of independence of tests). Alternatively, if subsequent test operations do not rely on the internal state of the *Test Evaluator*, a single instance can be used for a group of tests, potentially saving time in most scenarios.

2) Test case generation - the *Test Evaluator* instance receives the corresponding *Test case description* data. This description guides the test generator in creating specific values to test the student class. A test generator provides parameters in the form of a list or an object (*Generated test case data*), which are usually used as input to the constructor of the class under test. This test generator can be universal, generating random numbers, arrays, strings, etc., or its default functionality can be extended (for example, by inheriting from the universal generator). In this way, it can be tailored for specific tasks or groups of tasks.
3) Instantiating tested classes - the *Test Evaluator* contains code that defines a distinct sequence of operations for each test. Typically, the initial operation involves instantiating both the student and reference classes. Depending on the test format, the implicit (parameter less) constructor or the constructor specified in the assignment is used. The input parameters for these constructors are parameters prepared by the *Test Generator*.

students agreed with the statement that they are comfortable with an evaluation method based on checking the outputs of the methods (Q2).

- The new method of evaluation, based on the creation of a detailed evaluation of individual methods, was comprehensible to 74% of students at first glance (Q3).
- A significant majority, representing 79% of the partici- pants, agreed that the new output format was generally suitable for verifying the correctness of classes (Q4).
- However, only 55% of students report that with this form of feedback, they were able to identify errors more quickly and accurately compared to their experience in previous courses (Q5).
- Interestingly, 29% of students reported not noticing any change in the feedback format.

To illustrate the assessment and the situation, representa- tive statements of students are given. The positive feedback primarily centres around the ability to identify errors more rapidly:

- *‘‘It is clearly visible in which method there is an* *error, so there is no need to search the program as long as the other parts are correct.''*

- *''Easier debugging and troubleshooting, better clarity.''*
- *''It's easier to find out in which specific method I have an error; in the case of random values, I have more confidence that I wrote the code correctly and not just hit the output that the system wanted in a particular case.''*
- Conversely, some students expressed negative feelings rooted in their familiarity with the I/O-based assessment approach:
- *''I miss being able to get step-by-step feedback. Now, I must finish all constructors and methods before getting feedback.''*
- *''I found it challenging to identify bugs in the code because I had to write all the necessary methods to get the code to run.''*

The results of the survey are in line with the results of the previous research [102] conducted in the academic year 2019/2020. In this research, 69% of students found that automated assessments helped them understand new content, and 77% believed that automated assessments were effective for practising the content and satisfying for them.

V. DISCUSSION

Based on the principles of individual types of testing (black-, white-, grey-box), an integrating solution ensuring source code verification was designed, verified, and presented. In the module of object-oriented programming, which is the focus of this study, the solution is a tool for evaluating a class or a group of classes that meet the assignment requirements with their attributes and methods.

The basic concept and difference from industrial testing lie in the assumption that an object-oriented program- ming assignment always comes with an author's solution. By default, the author's solution exists to present a model solution to the student or is created by the author to verify the correctness and clarity of the assignment. The fundamental difference between the presented approach and standard automated testing is that a class of students is tested against the results of a reference class and not against specific predefined values.

The reference solution must meet the requirements of the assignment and pass all validation tests. As a result, its outputs (or the results of its methods) can be used as references for testing student classes. This approach eliminates the need to pre-define test parameter values; instead, these values can be generated continuously and randomly for each test.

Based on the existence of two classes that should present the same interface, comparing their behaviour (including outputs) is a matter of basic programming language mecha- nisms. Evaluating classes is easy due to the behaviour of their instances, and no specialized mechanism is needed in Java.

Comparing class behaviour reflects the results of indi- vidual methods. Access to the status of created instances represented by attributes is possible anytime. The evaluation procedure is based on comparing the results of the methods or their sequences between the tested and the reference class. An alternative or additional way is to compare attributes obtained from getters or summarise outputs from some method (appropriately defined already in the assignment).

To create input parameters, it is possible to use random generation of numerical or textual variables or their lists represented by arrays, objects, or other data structures.

The effectiveness of the presented solution lies in the coverage of several elements, each of which leads to efficiency in a specific area:

- Reducing the effort of the content creator for test preparation. Reducing the effort of the content creator is a key benefit of the whole concept. Investing effort
- in creating a reference solution is necessary, but the need to laboriously identify suitable inputs and search for their correctly corresponding outputs is eliminated.
- Minimizing dependencieson other libraries or software provides an opportunity for a durable solution that does not require frequent updates due to vulnerabilities or new library versions. Since the evaluation code does not use any specialized

libraries, it loses in versatility but gains in longevity.

- Minimizing requirements for transmission capacityis achieved because data transmission from the data server to the execution server is reduced to the transmission of formulas according to which test cases are created. There is also no need to transfer support libraries (xUnit).
- Cheating prevention was one of the basic requirements when designing the concept. It is based on the random generation of inputs and corresponding outputs. Although there may be a situation where a student's solution is accepted by the system even though it is incorrect, this situation is less likely and certainly less consequential than in the case of several predefined static tests. If it were incorrect, it would affect the assessment of all students.
- Student-friendly communication involves enriching information about expected and obtained outputs typical of black-box testing. Although this information may be part of the evaluation report, it must be supplemented with additional information. The first enrichment is the definition of the name of the test, which can correspond to the name of the tested method or a short description (e.g., constructor - boundary values). The optional part of the design is the supplementary feedback – if the authors anticipate or have experience with the most common mistakes students make in a specific test case, they can provide them with precisely targeted feedback.

The most important and intensive task in model implemen- tation is undoubtedly creating a test case. Randomness is a critical factor in this process, primarily to prevent cheating and speed up the generation of test cases. Strategies such as using templates for different data types or designing a simple language to specify data structure types and ranges can be adopted to increase the efficiency of creating input parameters (different approaches have been presented for different data types, such as numbers, texts, and arrays).

In addition, it is possible to generate invalid or completely random inputs using a fuzzy testing technique.

It should be noted that although the presented procedure provides an effective tool for creating assignments and testing the created programs, the application of random tasks in creating inputs requires a committed approach from the creator. A typical example is finding a random string in a list of random strings. The test case author must prepare the generation of input values so that the specified string should be found, at least in some cases. Otherwise, the student only needs a simple program listing the hard-coded output ''false'' or ''not found''.

Although the described solution was Java-oriented, the same approach can be used in Python, C, and PHP, or more generally, in any language that supports object-oriented programming.

For each test case and tested method, the student obtains information on whether the given method, constructor or output corresponds to the expected values, and for which values the test was successful or unsuccessful. Following the white and grey-box testing techniques, it is desirable to apply appropriate techniques to the greatest extent possible to achieve maximum code coverage and identify all problematic types of inputs.

Although solutions applying the presented methods are available, their disadvantage is robustness and complexity. The proposed solution is simple and does not require the implementation of additional libraries on the client or server side. In addition, it is independent of framework updates and relatively quickly adopted by creators of assignments. The next advantage is the transfer of data between the system intended for writing code and the execution server - it is only a few kilobyte files with the code responsible for running the tests.

The presented concept has its disadvantages, which can be summarized as follows:

- Despite the simplicity of the concept, it requires an advanced knowledge of the relevant programming language from the creator of the test.
- Each task or class validation requires the creation of a unique code. Although many checks are routine and often similar, no tool is currently available to create templates and speed up the creation of assessment classes. The creation of

evaluation classes is, therefore, time-consuming.

- Running and evaluating code that does not contain tested methods is impossible in Java. In other words, evaluating a partial solution is impossible because the methods must at least have a defined header. This limitation is due to pre-run code checks and general language safety rules. In contrast, in some other interpreted (and even some compiled) languages, the existence of a method is checked only when code is executed. This approach allows the evaluation of the program even in the absence of a corresponding method.

The presented solution was applied during four years of a university course in Java programming. One year took place in LMS Moodle using the Virtual Programming Lab module and three years in the educational system PRISCILLA. The students accepted the solution because it provided more detailed feedback than introductory courses using the I/O approach. Moreover, although this approach placed high demands on content creators in the initial tasks, they already appreciated the speed of their creation after getting better familiar with the principles.

CONCLUSION

Teaching programming is currently among the topics of interest to researchers because, despite advances in the field, the labour market is still experiencing a shortage of programmers. The preparation of tasks in educational environments can take several forms, while the goal and the expected current level of knowledge and skills of educators must be considered. In the case of introductory exercises and the creation of simple examples to familiarise students with the elements of a programming language, it is possible to consider various supporting algorithms and static analyses of the code before sending it for I/O analysis. After the movement to learning the principles of object-oriented programming, it is possible and probably appropriate to move code correctness verification to the area of unit testing.

By default, automated tests cannot be used universally with the same template for all assignment types. It is not even desirable in preparing tests used in teaching programming. The teacher or test creator is expected to adapt each test to the task. Although this approach is laborious, due to the specific preparation of each task, an explanation of errors typical for the given task can be integrated into the test. In more demanding cases, a solution guide can also be integrated.

The advantage of the presented solution is that it allows testing the class as a whole and the interconnection of several classes, verification of the relationship of inheritance and/or polymorphism, and even a special feature of the Java language – static.

Future work involves improving the presented solution in various ways. One promising approach is to modularize the system into separate layers: an input description layer (containing rules for generating inputs), a process description layer (controlling the sequence of methods execution), and a code evaluation layer (for comparing method results).

This division opens possibilities for designing templates following orthogonal array testing. These templates can streamline and automate the input generation process for individual or group parameters. A natural next step is to extend the solution from Java to other programming languages, such as Python and C/C++. The challenge lies in fine-tuning the concept to enable practical and student-friendly verification of classes and their methods while removing non-standard restrictions when working on assignments in the student's part of the environment. In addition, there is an opportunity to optimize human resources for content preparation. In testing-oriented sub- jects, it is possible to consider the inclusion of a thematic unit, where students with advanced skills would develop an evaluation code for specific tasks. This code could later be used in object-oriented programming subjects as the testing classes.

REFERENCES

[1] A. Zarifhonarvar, ''Economics of ChatGPT: A labor market view on the occupational impact of artificial intelligence,'' *SSRN Electron. J.*, pp. 1–31, Apr. 2023, doi: 10.2139/ssrn.4350925.

[2] S. Biswas, ''Role of ChatGPT in computer programming,'' *Mesopotamian J. Comput. Sci.*, vol. 2023, pp. 8–16, Jan. 2023, doi:

10.58496/mjcsc/2023/002.

[3] R. Yilmaz and F. G. K. Yilmaz, ''The effect of generative artificial intelligence (AI)-based tool use on students' computational thinking skills, programming self-efficacy and motivation,'' *Comput. Educ., Artif. Intell.*, vol. 4, 2023, Art. no. 100147, doi: 10.1016/j.caeai.2023.100147.

[4] X. M. Wang, G. J. Hwang, Z. Y. Liang, and H. Y. Wang, ''Enhancing stu- dents' computer programming performances, critical thinking awareness and attitudes towards programming: An online peer-assessment attempt,'' *Educ. Technol. Soc.*, vol. 20, no. 4, pp. 58–68, 2017.

[5] J. Liu, M. Sun, Y. Dong, F. Xu, X. Sun, and Y. Zhou, ''The mediating effect of creativity on the relationship between mathematic achievement and programming self-efficacy,'' *Frontiers Psychol.*, vol. 12, pp. 1–10, Jan. 2022, doi: 10.3389/fpsyg.2021.772093.

[6] A. Nguyen, C. Piech, J. Huang, and L. Guibas, ''Codewebs: Scalable homework search for massive open online programming courses,'' in *Proc. 23rd Int. Conf. World Wide Web*, Apr. 2014, pp. 491–502, doi: 10.1145/2566486.2568023.

[7] S. Djenic and J. Mitic, ''Teaching strategies and methods in modern environments for learning of programming,'' in *Proc. 14th Int. Conf. Cognition Explor. Learn. Digit. Age*, Oct. 2017, pp. 189–196.

[8] L. Cheng, A. D. Ritzhaupt, and P. Antonenko, ''Effects of the flipped classroom instructional strategy on students' learning outcomes: A meta-analysis,'' *Educ. Technol. Res. Develop.*, vol. 67, no. 4, pp. 793–824, 2019, doi: 10.1007/s11423-018-9633-7.

[9] H. Keuning, J. Jeuring, and B. Heeren, ''A systematic literature review of automated feedback generation for programming exercises,'' *ACM Trans. Comput. Educ.*, vol. 19, no. 1, pp. 1–43, Mar. 2019, doi: 10.1145/3231711.

[10] J. Skalka, M. Drlík, J. Obonya, and M. Cápay, ''Architecture proposal for micro-learning application for learning and teaching programming courses,'' in *Proc. IEEE Global Eng. Educ. Conf. (EDUCON)*, Apr. 2020, pp. 980–987, doi: 10.1109/EDUCON45650.2020.9125407.

[11] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, ''Towards practical programming exercises and automated assessment in massive open online courses,'' in *Proc. IEEE Int. Conf. Teach., Assessment, Learn. Eng. (TALE),* Dec. 2015, pp. 23–30, doi: 10.1109/TALE.2015.7386010.

[12] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, ''Pre- liminary results on using static analysis tools for software inspection,'' in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, 2004, pp. 429–439, doi: 10.1109/ISSRE.2004.30.

[13] T. Delev and D. Gjorgjevikj, ''Static analysis of source code written by novice programmers,'' in *Proc. IEEE Global Eng. Educ. Conf. (EDUCON)*, Apr. 2017, pp. 825–830, doi: 10.1109/ EDUCON.2017.7942942.

[14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, ''Why don't software developers use static analysis tools to find bugs?'' in *Proc. 35th Int. Conf. Softw. Eng. (ICSE),* May 2013, pp. 672–681, doi: 10.1109/ICSE.2013.6606613.

[15] M. Nouman, U. Pervez, O. Hasan, and K. Saghar, ''Software test- ing: A survey and tutorial on white and black-box testing of C/C++ programs,'' in *Proc. IEEE Region Symp. (TENSYMP),* May 2016, pp. 225–230, doi: 10.1109/TENCONSpring.2016.7519409.

[16] S. Nidhra, ''Black box and white box testing techniques—A literature review,'' *Int. J. Embedded Syst. Appl.*, vol. 2, no. 2, pp. 29–50, Jun. 2012, doi: 10.5121/ijesa.2012.2204.

[17] H. Liu and H. B. K. Tan, ''Covering code behavior on input validation in functional testing,'' *Inf. Softw. Technol.*, vol. 51, no. 2, pp. 546–553, Feb. 2009, doi: 10.1016/j.infsof.2008.07.001.

[18] K. M. Ehmer and F. Khan, ''A comparative study of white box, black box and grey box testing techniques,'' *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 6, pp. 12–15, 2012.

[19] A. Verma, A. Khatana, and S. Chaudhary, ''A comparative study of black box testing and white

box testing,'' *Int. J. Comput. Sci. Eng.*, vol. 5, no. 12, pp. 301–304, Dec. 2017.

[20] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, ''Fuzz testing in practice: Obstacles and solutions,'' in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 562–566, doi: 10.1109/SANER.2018.8330260.

[21] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, ''Evaluating fuzz testing,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2123–2138, doi: 10.1145/3243734.3243804.

[22] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, ''Fuzzing: A survey for roadmap,'' *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–36, Jan. 2022, doi: 10.1145/3512345.

[23] S. C. Reid, ''An empirical analysis of equivalence partitioning, boundary value analysis and random testing,'' in *Proc. 4th Int. Softw. Metrics Symp.*, 1997, pp. 64–73, doi: 10.1109/metric.1997.637166.

[24] W.-L. Huang and J. Peleska, ''Complete model-based equivalence class testing,'' *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 3, pp. 265–283, Jun. 2016, doi: 10.1007/s10009-014-0356-8.

[25] J. Zhang, Z. Zhang, and F. Ma, ''Introduction to Combinatorial Testing,'' in *Automatic Generation of Combinatorial Test Data* (SpringerBriefs in Computer Science). Berlin, Germany: Springer, 2014, pp. 1 –16, doi: 10.1007/978-3-662-43429-1_1.

[26] D. K. Ufuktepe, T. Ayav, and F. Belli, ''Test input generation from cause– effect graphs,'' *Softw. Qual. J.*, vol. 29, no. 4, pp. 733–782, Dec. 2021, doi: 10.1007/s11219-021-09560-3.

[27] K. Nursimulu and R. Probert, ''Cause-effect graphing analysis and vali- dation of requirements,'' in *Proc. Conf. Centre Adv. Stud. Collaborative Res.*, 1995, pp. 1–16.

[28] E. Krupalija, E. Cogo, Š. Bećirović, I. Prazina, and I. Bešić, ''Cause- effect graphing technique: A survey of available approaches and algorithms,'' in *Proc. IEEE/ACIS 23rd Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Dec. 2022, pp. 162–167, doi: 10.1109/SNPD54884.2022.10051799.

[29] A. R. Mahlous, A. Zarrad, and T. Alotaibi, ''State transition testing approach for ad hoc networks using ant colony optimization,'' *Int.*

[30] *J. Adv. Comput. Sci. Appl.*, vol. 9, no. 6, pp. 146–156, 2018, doi: 10.14569/IJACSA.2018.090621.