# A System-Level Testing Framework for Automated Assessment of Programming Assignments Allowing Students Object-Oriented Design Freedom

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

Nasser Giacaman
University of Auckland
Auckland, New Zealand
n.giacaman@auckland.ac.nz

***Abstract*—Automated assessment of programming assignments is essential in software engineering education, especially for large classes where manual grading is impractical. While static analysis can evaluate code style and syntax correctness, it cannot assess the functional correctness of students' implementations. Dynamic analysis through software testing can verify program behavior and provide automated feedback to students. However, traditional unit and integration tests often restrict students' design freedom by requiring predefined interfaces and method declarations.**

**In this paper, we present SysCLI, a novel testing framework for system-level testing of Java-based command-Line interface applications. SysCLI enables test suites that evaluate the functional correctness of students' implementations without limiting their design choices. We also share our experience using SysCLI in a second-year programming course at the University of Auckland, which focuses on object-oriented programming and design patterns and enrolls over 300 students each offering. Analysis of student assignments from 2023 and 2024 shows that SysCLI is effective in automating grading, allows software design flexibility, and provides actionable feedback to students. Our experience report offers valuable insights into assessing students' implementation of object-oriented concepts and design patterns.**

***Index Terms*—software testing for education, automated program assessment, system-level testing, software engineering education, Java programming, assignment-driven marking**

## I. Introduction

**Automated program assessment** [1]–[4] is essential in software engineering education [5], especially given the large class sizes in modern programming courses. Manually evaluating assignments for hundreds of students is impractical due to the significant effort required and the difficulties maintaining consistency and fairness in grading—especially when multiple graders are involved.

Existing approaches for automated grading program assignments generally use either static or dynamic analysis [6]. Static analysis focuses on aspects like code style [7] and syntax correctness. While it is useful for assessing design and structural properties, it cannot evaluate whether a program assignment's logic is semantically correct and behaves as intended [6].

In contrast, dynamic analysis, particularly **software testing** allows for evaluating assignments' behaviors by running code with various inputs and comparing the outputs to expected results. *Testing helps verify the student assignment correctness beyond what static analysis can achieve* [6], [8]. In addition, by knowing which tests pass or fail, students can receive automated feedback on the issues and limitations of their implementation [9].

However, developing test cases to grade student assignments raises the question: *"What type of test cases should an instructor use?"*

One solution is to implement **unit or integration tests** that directly invoke the methods implemented by students, and check the results against expected outcomes. Such tests require knowing the implementation details of the code being tested. If students were given the freedom in designing their classes and methods, there would need to be dedicated test cases that accommodate each student's individual implementation—it is impractical for the instructor to write dedicated test cases for each student. *Instead, a consistent test suite applicable across all student submissions is necessary*. In this scenario, instructors must provide the implementation design (classes and methods). Typically, students are either provided with starter code in the form of interfaces or empty method stubs adhering to the design, or provided with test cases that reveal the expected design, as these are the classes and methods that the unit/integration test cases expect. Providing the implementation design to students is particularly detrimental in Object-Oriented Programming (OOP) education. Students should be given the opportunity to design their own classes and methods, fostering a deeper understanding of software design. Specifically, when the course covers design patterns, providing stub methods and classes would reveal how to apply those patterns.

**System-level testing** [10] offers a better solution by allowing tests to focus on the input and output of the program without specifying implementation details. This can be achieved by invoking the program's `main` function with various inputs and checking the expected outputs. This approach enables students to retain the freedom to design their software in a way that aligns with their understanding, while still ensuring that their implementations are tested for correctness. However, writing system tests for program assignments can be challenging.

979-8-3315-0814-2/25 © 2025 IEEE

ICST 2025, Naples, Italy
Education Track

In particular, larger, long-term assignments have proven to be more effective for software engineering education compared to numerous smaller tasks [11]–[13]. They help students gain a deeper understanding of system architecture and design patterns [14]. Larger projects also provide a more realistic experience of software development [11]–[13]. **Command-Line Interface (CLI)** programs are particularly suitable for larger assignments of introductory programming courses [15], [16]. A CLI interface allows implementing multiple commands and their interactions. Unlike non-interactive command-line tools that handle a single input and output, CLI programs simulate real-world software scenarios, requiring students to develop more complex logic and manage program states, thus improving their understanding of object-oriented software design. Compare to Graphical User Interface (GUI) applications, interactive CLI programs enable students to focus on core functionality without the additional burden of designing a UI. Several independent studies agree that CLI-based assignments for beginner students are pedagogically superior to other types of applications [15]–[18].

*Implementing system-level test cases for interactive CLI applications using standard testing framework is challenging*. Indeed, with CLI applications we cannot simply invoke the `main` function passing some inputs as arguments. This is because the input of an interactive CLI program is not a list of arguments, but rather involves a sequence of user interactions. Despite the existence of libraries for implementing system-level tests for CLI applications in PYTHON [19] and BASH [20], there is no dedicated testing framework for Java-based CLI applications. This gap motivated us to design a new testing framework for JAVA CLI applications (called **SYSCLI**), which we used to create system-level tests to automatically grade and provide feedback on CLI-based assignments.

In this paper, we present an **experience report** on using our framework over two years (2023 and 2024) in a second-year programming course at the University of Auckland, which enrolls over 300 students in each offering. The course introduces students to OOP concepts and JAVA. It includes two large assignments involving CLI applications, totaling approximately 1,200 student solutions over the two years we analyzed. Notably, we provide students with half of the SYSCLI test cases that we will use for assessing them. This enables an alternative, simplified form of Test-Driven Development (TDD) [21], where students write code to pass the test cases.

We evaluated SYSCLI with two main research objectives: (i) to assess the effectiveness of the test cases in evaluating student assignments, and (ii) to determine whether the framework truly allows students the freedom to implement object-oriented designs. Our results show that the SYSCLI test suite achieves high code coverage (averaging over 90%) and mutation coverage (averaging over 80%) on student solutions. Additionally, our system-level testing approach enabled a noticeable variance in students' object-oriented designs, reflected in differences in the number of classes/interfaces, methods, and fields.

In summary, this paper makes the following contributions:

- A novel testing framework SYSCLI for interactive Java-based CLI applications.
- An experience report on using the framework in a second-year introductory programming course.
- An evaluation of the framework's effectiveness in ensuring program correctness while allowing students' design freedom.
- Insights into the use of system testing in educational settings for automated assessment and feedback of students' assignments.
- We publicly release SYSCLI [22] for use by other software educators and developers, with the hope that it will also encourage future work in this area.

The remainder of this paper is organized as follows: Section II introduces SYSCLI using a running example. Section III outlines the course context where we used SYSCLI for assessing and provide feedback on students' assignments. Section IV presents the experimental evaluation conducted to address the research objectives. Section V gives our reflections on the framework's usage. Section VI reviews related work, and finally, Section VII concludes the paper, highlighting potential directions for future work.

## II. SYSCLI

This section presents our testing framework using a running example.

### A. Running Example

To illustrate how SYSCLI works, we present a case study involving a CLI-based assignments we prepared, for which we developed test cases using our framework. This assignment tasks students with implementing a basic insurance management system. Listing 1 shows the program includes a set of commands. The assignment has ten commands, among which `HELP` and `EXIT` are already implemented for students. The `HELP` command displays the available commands, while the `EXIT` command terminates the application. Students are required to implement the logic for the remaining eight commands. The initial version of the code already implements the overall CLI mechanics; therefore, students only need to focus on implementing the core logic associated with the additional commands. In other words, the initial codebase comes equipped with the CLI shell architecture, which sets up the interface for users to interact. This infrastructure includes parsing commands, ensuring the correct number of arguments are provided, and managing the execution flow of commands.

It is important to highlight that this assignment is intended for second-year engineering students who are just beginning to learn object-oriented programming and Java. Although the assignment mentions a database, students are only required to simulate this functionality by storing information in memory (for example, in a `List`). This approach means that there is no

```
PRINT_DB        [no args]       Print the entire insurance database
CREATE_PROFILE  [2 arguments]   Create a new client profile <USERNAME> <AGE>
LOAD_PROFILE    [1 arguments]   Load the specified profile <USERNAME>
UNLOAD_PROFILE  [no args]       Unload the currently-loaded profile
DELETE_PROFILE  [1 arguments]   Delete the specified profile <USERNAME> from the database
POLICY_HOME     [no args]       Create a new home policy for the currently-loaded profile
POLICY_CAR      [no args]       Create a new car policy for the currently-loaded profile
POLICY_LIFE     [no args]       Create a new life policy for the currently-loaded profile
HELP            [no args]       Print usage
EXIT            [no args]       Exit the application

insurance system>
```

Listing 1: Commands for an assignment used as a running example. This is the output of the HELP command, listing the commands that provide functionality for managing an insurance database. It includes creating, loading, unloading, and deleting client profiles, as well as generating different insurance policies. The final line reflects the system prompt for students to interact.

persistence across application runs; each time the application is launched, it starts from a blank state. This design is crucial for our system-level testing framework, as each test case represents an independent interaction with the CLI application (i.e., a sequence of commands and their arguments) that should always begin from a clean state. This requirement is vital to avoid test flakiness [23] due to order dependency, allowing each test case to execute in isolation from the others.

Some commands within the assignment require arguments, while others do not. For instance, the `PRINT_DB` command operates without any arguments, whereas the `CREATE_PROFILE` command requires two specific arguments: the username and the age of the new client. Additionally, some commands, like `POLICY_HOME`, `POLICY_CAR`, and `POLICY_LIFE`, will prompt the user with questions to gather further information needed for creating the insurance policies.

An example user interaction with the CLI in interactive mode on a completed assignment solution is shown in Listing 2.

```
insurance system> CREATE_PROFILE John 23
  New profile created for John with age 23.
insurance system> PRINT_DB
  Database has 1 profile:
  1: John, 23, 0 policies for a total of $0
insurance system>
```

Listing 2: Example of interaction with the solution of our CLI-based assignment

In this interactive scenario, when `CREATE_PROFILE` is executed with the arguments `John` and `23`, the program should output a confirmation message. If the user subsequently executes the `PRINT_DB` command, the program indicates that there is now one profile and displays the relevant details.

Listing 3 shows an example of one of the many test cases that we used to grade students for this assignment. This test case verifies the interaction described above using SysCLI.

The static class `Task1` contains all the test cases for grading the first task of the assignment. This class extends `SysCliTest` (line 1, Listing 3), which implements our framework. By doing so, `Task1` inherits all the functionalities

```
1  public static class Task1 extends SysCliTest {
2
3  public Task1() {
4   super(Main.class);
5  }
6
7  @Test
8  public void T101_add_one_client_with_info() {
9   runCommands("CREATE_PROFILE John 23", ↩
        "PRINT_DB");
10  assertContains("New profile created for John ↩
        with age 23.");
11  assertContains("Database has 1 profile:");
12  assertContains("1: John, 23");
13  assertDoesNotContain("Database has 0 profiles");
14  }
15 }
```

Listing 3: Example of a test case implemented with SysCLI. The test verifies that a profile is created correctly.

we have developed. In particular, the `Task1` class gains access to the methods and properties defined in the `SysCliTest` class, providing common functionality for CLI testing, such as managing input/output streams, executing commands, and asserting the application's output.

The constructor of the `Task1` class calls the constructor of its superclass, `SysCliTest`, passing `Main.class` as an argument (line 4, Listing 3). `Main.class` serves as the entry point of the CLI-based application.

The `@Test` annotation indicates that the `T101_add_one_client_with_info` method is a JUnit test method (line 7, Listing 3). This enables SysCLI test cases to be executed by the JUnit testing framework.

The method invocation `runCommands` (line 9, Listing 3) prompts the CLI-based application with two commands in sequence: `CREATE_PROFILE John 23`, which creates a new profile for a client named John who is 23 years old, and `PRINT_DB`, which displays the content of the database. Note that each argument of `runCommands` is a single command; `CREATE_PROFILE John 23` has three literals because the command requires two arguments. You can think of each argument of `runCommands` as representing a command (and its arguments, if any) that the user inputs before hitting `Enter`.

TABLE I: Public Interface of the `SysCliTest` Class

| Method | Description |
|---|---|
| `CliTest(Class<?> mainClass)` | Constructor that initializes the class with the main class of the CLI-application. |
| `void setUp()` | Sets up the testing environment by redirecting output streams. |
| `void tearDown()` | Restores the `System` output streams and prints captured output after tests. |
| `void runCommands(String... commands)` | Executes a series of commands intercepting the `System.in` stream. |
| `void assertContains(String s)` | Asserts that the specified string is present in the output stream. |
| `void assertDoesNotContain(String s)` | Asserts that the specified string is not present in the output stream. |

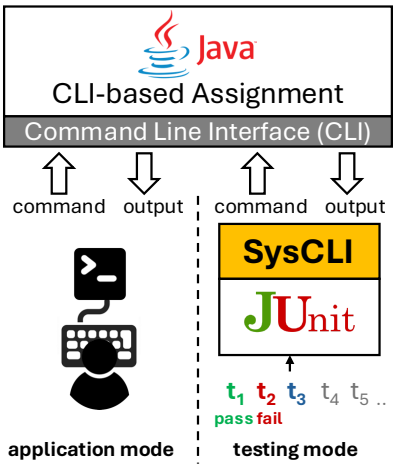


Fig. 1: Logical architecture of SysCLI. The left side is the interactive application mode with the user, while the right side is testing mode with SysCLI.

The test case contains four assertions:

1) `assertContains("New profile created for John with age 23.");` This assertion checks whether the output of the CLI contains a message confirming the creation of John's profile.
2) `assertContains("Database has 1 profile:");` This assertion checks that the output indicates there is indeed one profile in the database.
3) `assertContains(" 1: John, 23");` This assertion verifies that the output lists John as the first profile, confirming that the profile was created (and stored) correctly. Note that the framework allows checking only a portion of the expected output to provide some flexibility; indeed, the full output was `1: John, 23, 0 policies for a total of $0`, but this test case is checking for the simplest functionality.
4) `assertDoesNotContain("Database has 0 profiles");` This assertion ensures that the output does not indicate that there are zero profiles in the database. For example, if the order of the two commands is inverted, `PRINT_DB` will print `Database has 0 profiles`. This design prevents students from writing all possible outputs for each command to pass the tests.

### B. Framework Implementation

Table I shows the Application Public Interface (API) of our framework, while Figure 1 overviews its logical architecture. The left side of the figure illustrates the typical usage of the CLI-based assignment, where the program waits for user commands entered via the console, and the user reads the output printed in the console. The right side shows the testing mode with our framework. In this mode, SysCLI automatically supplies the commands specified by the test case when the program expects user input. It then captures the output and applies assertion oracles to determine whether the test passes or fails.

We implemented SysCLI on top of JUnit 4[1]. Specifically, the methods `assertContains` and `assertDoesNotContain` rely on `Assert.fail()` of JUnit to trigger a test failure when the assertion oracles are not satisfied. This design ensures that the tests integrate seamlessly with build automation systems such as Gradle and Maven, as well as Java Integrated Development Environments (IDEs). This is because SysCLI's tests can be executed as JUnit tests.

The `setUp` and `tearDown` methods are annotated with the JUnit `@Before` and `@After` annotations, respectively. This means that `setUp` automatically executes before each test, while `tearDown` afterward. The `setUp` method redirects the output and error streams from `System.out` and `System.err` to internal `PrintStream` variables. This allows SysCLI to capture the output of the application during execution. The `tearDown` method restores the `System.out` and `System.err` streams. It also prints the captured output to the console, providing students with more detailed information about the test execution in case of test failures.

The `runCommands` method accepts an arbitrary long list of Strings (i.e., the commands and arguments). It uses Java reflection to invoke the `main` method of the CLI application that is passed as a parameter in the class constructor (e.g., see Listing 3, line 4). It also uses reflection to automatically override the `java.util.Scanner` instance that the application uses to read input from the console. This allows SysCLI to emulate users typing the commands when the application prompts for user input. The commands are those specified in the `runCommands` method parameters.

[1]Our framework can be easily adapted for JUnit 5; we chose JUnit 4 due to its broader compatibility with existing projects.

For example, consider the scenario in Listing 3. Because `runCommands` invokes the `main` method with an internal `Scanner`, when the CLI application requests input from `System.in`, the internal `Scanner` supplies the specified inputs. In this example, when the CLI application prompts for user input, it immediately receives `CREATE_PROFILE John 23` from the internal `Scanner`. The next time the application waits for input, it receives `PRINT_DB`.

The `assertContains` and `assertDoesNotContain` methods ensure the test fails if the captured output does not contain the specified string, or, in the case of `assertDoesNotContain`, if the output unexpectedly includes the specified string. When an assertion fails, it prints a message to the console detailing the arguments of the assertion and explaining the reason for the failure (i.e., whether the specified output was present or absent). Additionally, a test will fail if the program crashes, enters an infinite loop, or falls into infinite recursion. To handle the last two cases, SysCLI uses JUnit `@Rule public Timeout timeout`, which automatically makes a test fail if it exceeds a specified duration (set to 10 seconds by default).

In a nutshell, what makes our framework effective is its seamless interaction with the CLI application. The application cannot distinguish between inputs provided by a user and those supplied by a test; it behaves as if a user is interacting with it. This process is fully automated and handled by our framework in the background.

## III. Experience Context

This section provides an overview of the course, its assignments, the process of creating test cases, as well as the assessment and feedback methods used for evaluating the assignments.

### A. Course Context

We applied SysCLI to the *SOFTENG281 Object-Oriented Programming*[2] course at the Faculty of Engineering, University of Auckland, New Zealand. SOFTENG281 is a second-year course focusing on object-oriented programming [24] and is mandatory for all students specializing in Software Engineering, Computer Systems Engineering, and Electrical and Electronics Engineering. These are three specializations offered within the four-year Bachelor of Engineering (Honours) degree.

Before specializing in the second year, students complete the same foundational courses in the first year of their degree, regardless of their intended specialization. *ENGGEN131 Introduction to Engineering Computation and Software Development*[3] is one such course, which teaches the fundamentals of programming using MATLAB and C, and serves as the prerequisite for SOFTENG281.

With a yearly enrollment of over 300 students, SOFTENG281 uses Java to introduce object-oriented programming concepts, as Java is used extensively in other courses within the Software Engineering degree, ensuring continuity in the students' learning experience. The course structure includes two six-week segments. The first segment introduces fundamental Java programming and basic OOP principles, such as methods, objects, encapsulation, inheritance, polymorphism, and abstract classes, along with control structures. In the second segment, students learn more advanced topics such as Java interfaces, design patterns, exception handling, and fundamental data structures. This segment also introduces graph traversal algorithms, such as depth-first and breadth-first searching.

We use a semi-flipped classroom approach, where hands-on coding sessions are prioritized during lectures, while students are expected to study the theoretical content beforehand [25], [26]. We consider this to be semi-flipped, because brief reviews of theoretical content are still conducted during class, but the focus is on practical coding exercises.

### B. Course Assignments

SOFTENG281 is designed as an assignment-focused course, aligning with research that highlights the effectiveness of large coding assignments in practical programming education [27]. Informed by prior research [28], we organized the course around two major programming assignments, which account for about 60% of the final grade for the course, while the remaining 40% is assessed through invigilated tests, where students solve smaller coding tasks within a controlled environment. **Assignment 1** (**A1**) covers the content of the first part of the course, while **Assignment 2** (**A2**) covers the content of the second part.

For the distribution and submission of both assignments, we use GitHub Classroom [29]. Each assignment comes with starter code that follows a consistent template across all assignments. The setup is a Maven project, which includes the basic CLI structure while deliberately excluding the internal logic. This setup allowing students to easily build, run, and test the application using simple Maven commands.

### C. Creating Assignment Test Cases

In developing a new assignment (and its associated test cases), we first craft the complete solution. We then execute the solution program in a terminal, identifying canonical and representative user interaction scenarios. By capturing these interactions, we create test cases that reflect typical user behavior. We translate the console interaction into the corresponding `runCommands` methods, accompanied by `assertContains` and `assertDoesNotContain` assertions. In particular, each test case emulates a single user interaction scenario.

After creating the complete set of test cases, we divide them into two categories: *visible* and *hidden* test cases. Half of the test cases are provided to students (*visible*), while the complete set (*visible* plus *hidden*) is used for grading the assignment. The visible test cases focus on the core functionalities of the application, providing students with a clearer understanding of the main requirements. The hidden test cases typically target

[2]https://courseoutline.auckland.ac.nz/dco/course/SOFTENG/281
[3]https://courseoutline.auckland.ac.nz/dco/course/ENGGEN/131

TABLE II: Summary of passing test cases in Assignments for 2023 and 2024.

| Year | # Students | Assignment | # SysCLI Tests | Students passing all tests | Passing Test Cases | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Mean | SD | Min | Q1 | Median | Q3 | Max |
| 2023 | 309 | A1 | 42 | 153 (50.0%) | 91.2% | 6.81 | 16.7% | 90.5% | 100.0% | 100.0% | 100.0% |
| | | A2 | 84 | 216 (70.0%) | 96.1% | 12.52 | 3.6% | 100.0% | 100.0% | 100.0% | 100.0% |
| 2024 | 302 | A1 | 75 | 117 (38.7%) | 87.9% | 12.23 | 4.0% | 86.7% | 94.7% | 96.0% | 100.0% |
| | | A2 | 66 | 164 (54.3%) | 95.5% | 7.68 | 9.1% | 98.5% | 100.0% | 100.0% | 100.0% |

edge cases, while also helping to ensure that students do not hard-code their solutions to pass only the visible tests.

When releasing each assignment, we include a detailed handout that outlines the program specifications. In addition to the handout, the SysCLI test cases provided to students serve as further concrete examples of expected interactions with the application. Research has shown that examples are highly effective in conveying requirements, and in this context, SysCLI test cases act as a form of executable specification [30].

Given the complexity of these assignments, which often require several weeks of effort, we divide each assignment into tasks, each representing a specific functionality of the application. We typically have three to six tasks[4], and often use these as checkpoints to ensure students are making progress. The test cases are organized according to these tasks, allowing students to focus on one aspect at a time (there is typically 10–20 test cases per task). This approach enables a simplified form of Test-Driven Development (TDD), where all tests initially fail, and students are required to implement code incrementally to make each test pass [31], [32]. Unlike standard TDD, students are not required to write the initial failing tests to guide their implementation. However, the presence of failing tests still helps them to effectively guide their development process.

The test suite is structured to support this incremental development process, as shown in Listing 4.

```
@SuiteClasses({
MainTest.Task1.class,
MainTest.Task2.class, // Uncomment for Task 2
// MainTest.Task3.class, // Uncomment for Task 3
})
```

Listing 4: Example of a SysCLI Test Suite divided by tasks

Students will uncomment specific test classes as they progress through the tasks, allowing them to verify their solutions step by step (only the uncommented Test classes will be executed by JUnit). This design not only helps students stay organized as they work through the assignment, but also provides them with a structured way to test their progress.

### D. Course Assessment and Feedback

After the deadline has passed, we grade the assignments using an automated Python script. This script clones the repositories from GitHub Classroom, replaces the `src/test/java` folder with a directory containing all the tests (both *visible* and *hidden*), executes `mvn test`, and reports which test cases have passed or failed. The grade for the assignment is proportional to the number of passing test cases. We also automatically evaluate Java code style using our GradeStyle tool [7]. Additionally, we assess whether students have implemented the required design patterns using an automated static analysis script that we developed to detect these patterns. Students receive an automated email with a detailed test report, indicating which test cases passed or failed. After grading, we release the complete set of test cases, allowing students to identify their failing tests and run/debug their applications to understand and correct their mistakes.

[4]2023-A1 has 3, 2023-A2 has 6, 2024-A1 has 3, and 2024-A2 has 5 tasks

## IV. Evaluation

This section presents a series of experiments that we conducted to evaluate two key aspects of our framework: (1) the effectiveness of the test cases in assessing student assignments, and (2) whether the framework truly allows students object-oriented design freedom.

### A. Testing Results

Table II summarizes the results for the passing test cases in the assignments for 2023 and 2024, the years in which we adopted SysCLI. Additionally, the table shows the number of students, the assignment identifier (A1 or A2), the total number of test cases (*visible* + *hidden*), the number of students passing all the tests, and statistical measures of the number of passing test cases. The results across both years and assignments indicate that the majority of students passed a substantial number of test cases. In fact, the average percentage of passing test cases across assignments and years ranges from 91.2% to 96.1%. This suggests that the assignment design was effective in enabling students to produce correct code that could often also pass the hidden test cases.

### B. Evaluating Test Effectiveness

To evaluate the effectiveness of SysCLI test cases across the different student assignments, we rely on three standard test adequacy metrics:

- *Line Coverage*: This metric measures the percentage of lines of code executed by the test suite. It helps to assess how thoroughly the assignment is exercised during testing.
- *Mutation Coverage*: This metric measures the percentage of artificially seeded faults (mutants) that are detected and killed by the test suite. It not only evaluates how well the test cases cover the code but also measures the

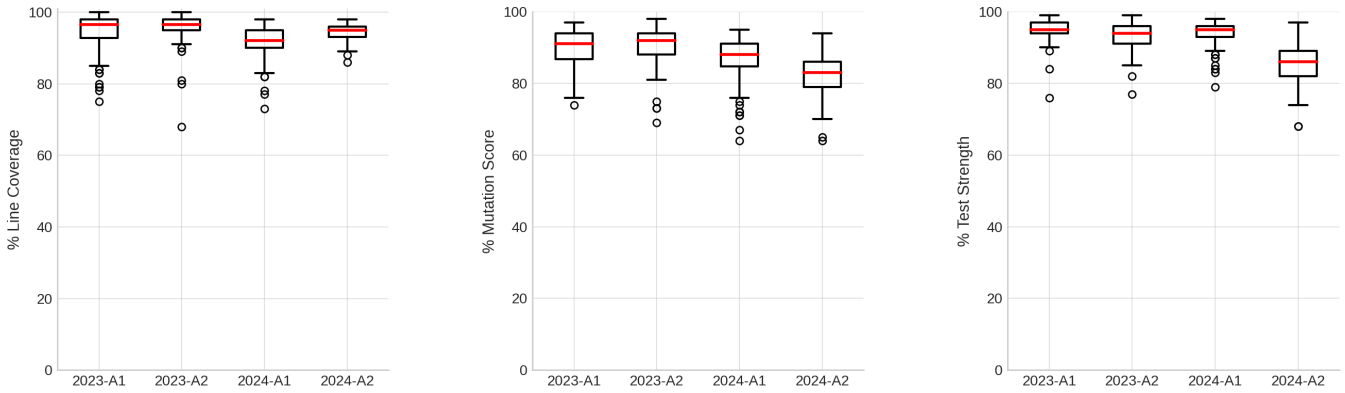


Fig. 2: Distribution of Line Coverage (left), Mutation Coverage (center), and Test Strength (right) for the assignment solutions that pass all the SysCLI test cases.

effectiveness of the assertions in catching seeded faults. The study of Clegg et al. shows that common mutants are suitable substitutes for students faults [33]. In particular, the study indicates that mutants capture the observed faulty behavior of students' solutions.

- *Test Strength*: This metric evaluates the proportion of mutants killed out of the total mutants that were actually covered by the test suite. It excludes mutants that survived due to lack of test coverage. This provides a clearer measure of how effective the assertions are in detecting faults, independent of coverage gaps.

We computed these metrics for only those student solutions that passed *all* the visible and hidden test cases. We made this choice for two reasons: First, it ensures consistency, as passing all test cases provides strong confidence that the solutions are correct and complete; comparing complete with incomplete assignments would be meaningless. Second, mutation testing requires a fully passing (green) test suite to yield valid results. As indicated in Table II, at least half of the students passed all the test cases in three out of four assignments.

We used the Maven plugin of PIT (v. 1.17.0)[5] to calculate the metrics. We excluded the classes that we provided to students for implementing the CLI interface (e.g., `Main.class`), as students were not allowed to modify these classes. We configured PIT with its default settings, which apply the "DEFAULTS" group of mutators[6]. The mutators used by PIT target low-level implementation faults, such as modifying conditional boundaries or negating integer values.

Figure 2 shows the box plot of the distribution of Line Coverage (left), Mutation Coverage (center), and Test Strength (right) for the assignment solutions that pass all the SysCLI test cases.

**Line Coverage** is generally high, with the average across the four assignments ranging from 92% (2024-A1) to 95.8% (2023-A2). This suggests that the test cases are adequate to test the assignments and that students did not over-engineer their solutions by adding extra features or behaviors not covered by the tests. Research has shown that TDD encourages lean implementations by reducing the likelihood of unnecessary code [32]. While there are some outliers, as shown in the box plot, they are relatively few.

**Mutation Coverage** is also generally high, with the average across the four assignments ranging from 82.1% (2024-A2) to 90.2% (2023-A2). It is important to note that the mutation score is calculated by applying the same test suite across different student implementations. Consequently, the number of mutants generated by PIT may vary across assignments due to differences in code structure and complexity, which can impact the mutation score. Nonetheless, a high mutation coverage indicates that the test cases are not only covering the code but are also effective at detecting seeded faults, demonstrating strong fault detection capabilities [34], [35].

**Test Strength** is high, with the average across the four assignments ranging from 85.3% (2024-A2) to 94.8% (2023-A1). As expected, because test strength measures the proportion of killed mutants relative only to the mutants that were covered by the tests, the results are generally higher than those for overall mutation coverage. This indicates that once the tests cover a specific piece of code, they are highly effective at detecting faults. The high test strength also suggests that the system-level assertions in the test cases are successful in uncovering low-level implementation errors.

As discussed in Section III, it is important to clarify that we implemented the test cases using a pure black-box approach, which is agnostic to specific implementation details (since we cannot predict all potential student implementations). This differs from a white-box approach aimed at maximizing coverage in the reference solution implemented by us. Despite this, our results show that it is possible to develop system-level tests in a black-box manner that are effective in assessing the correctness of various student solutions, regardless of implementation details (which vary across students' solutions) [36], [37].

[5]https://pitest.org/quickstart/maven/
[6]https://pitest.org/quickstart/mutators/

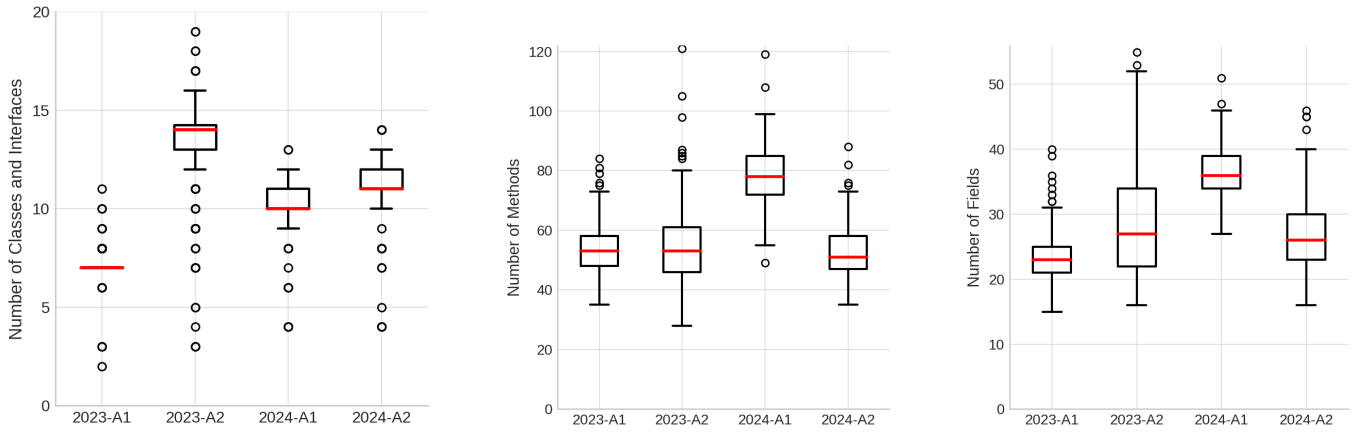


Fig. 3: Distribution of the number of classes and interfaces (left), number of methods (center), and number of class fields (right) for the assignment solutions that pass all the SysCLI test cases.

### C. Evaluating OO Design Freedom

In our second experiment, we analyzed the object-oriented characteristics of student solutions to evaluate the diversity of their OO design choices. While directly assessing OO design decisions is challenging, we used proxies such as the number of classes or interfaces created, the number of methods defined, and the number of fields declared. These metrics offer a broad understanding of the design structure without going into detailed implementation specifics. We intentionally avoided metrics like Lines of Code (LOC) and Cyclomatic Complexity (CC), as they focus more on the low-level implementation characteristics and do not offer meaningful insights into the design itself.

To compute the metrics we relied on the open-source Java code metrics calculator CK, implemented by Maurício Aniche [38]. Consistent to the computation of test adequacy, we chose to analyze only the student solutions that pass all test cases. This ensures a meaningful comparison, as we can be confident that these solutions are complete.

Figure 3 illustrates the distribution of the number of classes and interfaces (left), methods (center), and class fields (right).

The distribution of **the number of classes and interfaces** reflects the diversity of student solutions. The graph shows that A1 in both 2023 and 2024 has less variance in the number of classes and interfaces compared to A2. This is because A2 requires students to implement specific OO design patterns, which often result in a greater number of classes and encourage more modular and encapsulated designs. The higher variability observed in 2023-A2, as reflected by the Standard Deviation (SD) 2.60, suggests more diverse design choices among students.

The **number of methods** varies considerably across student solutions, indicating noticeable differences in design complexity. The SD ranges from 8.71 for 2023-A1 to 12.78 for 2023-A2.

The **number of fields** across solutions reflects differing approaches to class definitions. The SD ranges from 3.97 for 2024-A1 to 8.38 for 2023-A2, indicating varying levels of data encapsulation. The higher standard deviation in 2023-A2 suggests that some students opted for more attributes to represent additional class states, possibly influenced by the design patterns required in this assignment.

Overall, there is a noticeable variance in the distributions, which reflects the design freedom allowed by SysCLI system-level tests. This level of variability would not be possible with traditional unit/integration tests, as they invoke specific code, thereby constraining the design and limiting the range of possible implementations.

## V. Reflection and Lessons Learned

This section reflects on our experience using SysCLI, highlighting the main challenges we faced and what went well. It also presents a series of lessons learned, offering valuable advice for educators interested in using SysCLI to assess and provide feedback on student assignments.

### A. What Went Well

**Ease of writing tests:** Writing test cases with our framework is intuitive, as it simply involves specifying inputs and expected outputs of user interactions with the CLI. We found that an effective approach is to copy our console interactions with the assignment solutions (input and output), and then translate these into the `runCommands` methods accompanied by `assertContains` and `assertDoesNotContain` assertions to validate the outputs. Interestingly, we discovered that some students, without prior knowledge of software testing, were able to create their own system-level test cases using SysCLI, inferring what the hidden test cases might cover. The simplicity of SysCLI encouraged students to share their test cases with each other, helping them gain confidence in their implementations and better prepare for the hidden test cases.

**System-level testing vs. LLMs:** While LLMs demonstrate proficiency in tackling simple tasks typical of introductory courses, they face some difficulty with more complex tasks that demand higher levels of problem solving [39]. We have observed that SYSCLI test cases present a challenge for LLMs when generating solutions for our assignments. LLMs tend to struggle with solving larger assignments that require coordination across multiple classes and components. Although we have not yet rigorously evaluated LLMs on solving our assignments, we have noticed that CHATGPT performs poorly at completing our assignments when provided with SYSCLI's test cases. We conjecture that unit and integration tests often reveal detailed implementation hints, which could make it easier for LLMs to produce correct solutions. In contrast, SYSCLI tests focus solely on specifying command-level interactions, withholding all implementation details. This may increase the difficulty for LLMs in generating correct solutions.

**Design freedom and plagiarism detection:** Granting students design freedom in their assignments led to a wide range of student solutions, often employing very distinct design decisions. This diversity was reflected in the high variance of software metrics across the submissions (see Section IV). We have also found that such diversity across student submissions makes it extremely improbably and suspicious for two students to independently arrive at an identical OOP design solution. Prior to incorporating SYSCLI into our assignments, where students were required to follow the same implementation design prescribed by unit and integration tests, detecting plagiarism was very challenging due to the uniformity of expected solutions. We have found that SYSCLI not only facilitates creativity and critical thinking, but it also helped us to more effectively identify plagiarism cases.

### B. Challenges

**Maintaining the test cases:** One of the biggest challenges we face is the maintenance of SYSCLI test cases. Although writing these tests is simple, modifying them to align with frequent assignment changes can be challenging—especially when drafting a new assignment. Unlike standard unit or integration JUNIT tests, SYSCLI tests require careful review of all string arguments in both commands and assertions. When a test prescribes a complex sequence of interactions, it becomes long and includes multiple assertions, which can be difficult to update without extensive review. The presence of multiple assertions within a single test is often unavoidable in SYSCLI, as the interactions tested often involve a sequence of commands. Test cases containing multiple assertions are sometimes considered suffering from a "test smell", known as *Assert Roulette* [40]. Multiple assertions can make debugging more challenging, as developers may need to spend additional time identifying the specific cause of test failure. However, recent research by Panichella et al. [41] suggests that the impact of Assert Roulette on test readability and maintainability has reduced with the advancement of modern testing frameworks, which now provide feedback on individual assertion failures. SYSCLI also provides such a feedback (see Section II).

It is worth noting that, unlike real-world software projects, assignments in educational settings are not expected to evolve over time, so test cases generally do not require continuous updates. However, should assignments be revised before releasing to students, maintaining and updating the test cases to reflect these changes can be a delicate process. In such cases, it may be more efficient to discard the old test cases, and create new one by copying and pasting the updated CLI interactions obtained when running the revised solution.

### C. Lessons Learned

**Use meaningful test case names:** Descriptive test case names help students quickly understand the purpose of each test. They also provide immediate context when a test fails, making it easier for students to identify the specific functionality that is not working. Instead of JAVA's traditional *camelCase*, we use underscores in test names—a trend that enhances readability in longer, descriptive JUNIT test names [42].

**Start with simple tests:** To boost student confidence, we suggest beginning each assignment with very simple tests that students can pass easily. We also use `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` to ensure that test cases are executed in alphabetic order, as specified by an incremental identifier in the test names. For example, the prefix `T101` in Listing 3 specifies that this is the first test case of Task 1.

**Avoid creating test cases that are too similar:** Tests that are too similar can make a sudden jump in passing multiple tests with a single code change, which undermines the gradual progress intended by a test-driven approach. We also ask students to make a GIT commit each time they pass a new test case. This practice helps us identify plagiarism or suspicious cases where students make a single commit with a full solution, allowing us to observe more genuine, incremental progress.

**Ensuring robust test cases with teaching assistants:** To ensure test cases are correct, have teaching assistants complete the assignment using only the handout and visible test cases — simulating the student experience — before releasing it. Then run all tests (both visible and hidden) on their solutions to verify they pass. Any failing tests may indicate issues with the assignment instructions or inconsistencies in the hidden test cases. We have consistently found this approach helps identify problems before releasing the assignment to students.

**Using GitHub Copilot:** Using an IDE with the GITHUB COPILOT plugin greatly enhanced productivity in generating test cases. The plugin, which is free for educators [43], was helpful for brainstorming test scenarios and coding SYSCLI test cases. After creating a few initial tests, GITHUB COPILOT quickly grasped the assignment's interactive CLI nature. Often, simply providing a descriptive test method name was enough for the plugin to generate most of the test code. Alternatively, instructors could paste terminal output as a comment before the test case, allowing GITHUB COPILOT to translate it into the required SYSCLI format.

**Preventing hard-coded solutions by students:** To discourage students from hard-coding logic to pass specific visible test cases, we inform them that we will check for such attempts. We achieve this by modifying the visible test cases using simple find-and-replace methods (e.g., changing `"John"` to `"Jenny"` or `"23"` to `"25"`), while maintaining the original intent. This approach ensures that genuine solutions remain effective regardless of these changes and prevents students from undermining the assignment's integrity. It encourages them to develop robust solutions that work for a range of equivalent values—not just to pass the visible test cases, which should be viewed only as exemplars to guide them in their development.

**Discuss the assignment design in class:** After each assignment deadline, we held in-class discussions to review the instructor's solution. This helps students understand key software design decisions and recognize the qualities of a well-designed solution, supporting learning of OOP design [44]. We highlighted that multiple equally valid design solutions exist and encouraged students to share their ideas, fostering open dialogue on design practices. This approach presented various ways to solve the same problem and provided opportunities for peer learning and constructive feedback.

## VI. Related Work

This section discusses related work on SysCLI in assessing programming assignments and system-level testing frameworks.

**Automated program assessment** of student coding assignments has been widely studied, with several surveys that overview the field [1]–[6]. However, the automated assessment and feedback of CLI-based assignments remains underexplored, despite evidence suggesting that CLI-based assignments are highly effective for introductory software engineering courses [15]–[18]. Indeed, CLI-based tasks simulate real-world software usage scenarios offering practical benefits like simplicity and emphasis on core programming skills [15].

**System-Level testing frameworks** primarily focus on Web and GUI applications, aiming to simulate user interactions in GUI environments. Popular frameworks are Selenium [45] and Cypress [46], which automate web testing, and tools like AutoIt [47] and Sikuli [48], which are designed for GUI automation. These tools lack support for CLI applications, which involve text-based command input and output parsing. In contrast, SysCLI is a system-level testing framework for interactive CLI-based applications.

**Testing frameworks for CLI-based applications:** Expect [49] is a tool primarily designed for task automation of Bash applications. It enables users to automate CLI interactions by simulating command inputs and capturing program outputs. While Expect is effective for automating complex terminal workflows, it lacks structured test case definitions and built-in assertion mechanisms. In contrast, SysCLI allows us to define inputs and assertions in a structured manner, similar to how JUnit tests are defined. CliTest [19] and BATS [20] are frameworks for implementing system-level tests for Python and Bash CLI applications, respectively. Both CliTest and BATS offer a way to automate command-line interactions and verify outputs. However, both tools are limited to their respective languages and do not support Java-based CLI applications, making them not applicable in contexts where Java is the chosen language for CLI development. Moreover, *to the best of our knowledge, there have been no studies examining the use of system-level testing tools (like* CliTest *and* BATS*) for assessing student assignments.*

## VII. Conclusions and Future Work

This paper presented an experience report on using SysCLI, a novel framework for developing system-level test cases to assess and provide feedback on student assignments. Overall, our experience has been highly positive. Our evaluation demonstrates that SysCLI test cases are effective in assessing student assignments while also allowing students design freedom, which is essential for fostering a deeper understanding of object-oriented programming principles.

We plan to continue using SysCLI in our course and we encourage the software engineering education community to adopt it as well. There are several exciting future works for SysCLI. We now discuss the three most promising ones.

**Adding new assertion types:** Expanding assertion types would enhance SysCLI's test case expressiveness. For example, an `assertMatchesRegex(regex)` could verify output against regular expression patterns, useful for testing specific formats like dates. Another example is `assertCountString(string, n)`, which could check if a string appears a specified number of times in the output.

**Record and replay:** Testing frameworks for web and GUI applications often offer record-and-replay modes to automatically translate interactions into test cases. For example, Selenium [45] provides a plugin that records user interactions and generates JUnit test cases. We envision a similar feature for SysCLI, where a specific command (e.g., `START_RECORDING`) begins recording interactions, and `END_RECORDING` stops the recording and automatically generates a SysCLI test case. This could significantly accelerate test case creation for instructors and help students generate regression tests. Assertions like `assertContains` could be auto-generated from console output, and test names could be suggested by LLMs based on the test code and handout.

**Automated test generation using LLMs:** Automated test generation for programming assignments [50] has potential to further reduce instructor effort in defining test cases. We began exploring the idea of using a white-box approach to generate commands that maximize branch coverage of the instructor solution. However, as discussed in this paper, we believe that a black-box approach is preferable, as white-box-generated tests, while effective for coverage, may not align with canonical or meaningful usage of the application. Given our positive experience with GitHub Copilot, future research could explore whether LLMs can generate semantically meaningful tests based on the assignment handout.

## REFERENCES


[1] R. Romli, S. Sulaiman, and K. Z. Zamli, "Automatic programming assessment and test data generation a review on its approaches," in *2010 International symposium on information technology*, vol. 3. IEEE, 2010, pp. 1186–1192.

[2] S. Nayak, R. Agarwal, and S. K. Khatri, "Automated assessment tools for grading of programming assignments: A review," in *2022 International Conference on Computer Communication and Informatics (ICCCI)*. IEEE, 2022, pp. 1–4.

[3] H. Aldriye, A. Alkhalaf, and M. Alkhalaf, "Automated grading systems for programming assignments: A literature review," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 3, 2019.

[4] J. C. Caiza and J. M. Del Alamo, "Programming assignments automatic grading: review of tools and implementations," *INTED2013 Proceedings*, pp. 5691–5700, 2013.

[5] A. Luxton-Reilly, E. Tempero, N. Arachchilage, A. Chang, P. Denny, A. Fowler, N. Giacaman, I. Kontorovich, D. Lottridge, S. Manoharan, S. Sindhwani, P. Singh, U. Speidel, S. Stephen, V. Terragni, J. Whalley, B. Wuensche, and X. Ye, "Automated Assessment: Experiences From the Trenches," in *Proceedings of the Australasian Computing Education Conference*, 2023, pp. 1—-10.

[6] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Transactions on Computing Education (TOCE)*, vol. 19, no. 1, pp. 1–43, 2018.

[7] C. Iddon, N. Giacaman, and V. Terragni, "GradeStyle: GitHub-Integrated and Automated Assessment of Java Code Style," in *IEEE/ACM International Conference on Software Engineering, SEET track*, 2023, pp. 192–197.

[8] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 15–26.

[9] K. Buffardi and S. H. Edwards, "The role of automated feedback in supporting students learning computer science," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 2015, pp. 529–534.

[10] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. John Wiley & Sons, 2011.

[11] T. B. Hilburn and M. Towhidnejad, "Large projects as capstones: Benefits for software engineering students," *IEEE Transactions on Education*, vol. 50, no. 3, pp. 224–232, 2007.

[12] D. Parsons and M. Mentis, "Large-scale software engineering projects in education: Learning benefits and challenges," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. ACM, 2012, pp. 21–26.

[13] G. Schilling and M. Harris, "The impact of long-term projects on student learning in software engineering courses," in *Proceedings of the 2015 ASEE Annual Conference & Exposition*. ASEE, 2015, pp. 1–8.

[14] M. Fowler and A. Scott, "Experience report: Large project assignments in software engineering courses," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 2002, pp. 469–476.

[15] S. Rogerson and P. Biddle, "Cli vs. gui: Teaching programming fundamentals in an introductory course," in *Proceedings of the 2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2019, pp. 1–5.

[16] A. Gupta and N. Sharma, "Teaching introductory programming through command-line interfaces: A case study," *Journal of Computer Science Education*, vol. 30, no. 2, pp. 105–123, 2020.

[17] J. Foster and T. Rahman, "Using command-line programs to teach software development principles," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 128–133.

[18] R. Milliken and A. Smith, "Introducing command-line tools to first-year computing students," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2018, pp. 300–305.

[19] A. Jargas, "clitest: Command line interface tester," https://github.com/aureliojargas/clitest, 2024, accessed: 2024-10-26.

[20] B. C. Team, "Bats-core: Bash automated testing system," https://github.com/bats-core/bats-core, 2024, accessed: 2024-10-26.

[21] M. M. Hakan Erdogmus and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, 2005.

[22] V. Terragni and N. Giacaman, "Syscli: A system-level testing framework for cli applications," 2024, https://github.com/Digital-Educational-Engineering/syscli.

[23] V. Terragni, P. Salza, and F. Ferrucci, "A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests," in *42nd IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2020, pp. 69–72.

[24] N. Giacaman, P. Roop, and V. Terragni, "Evolving a Programming CS2 Course: A Decade-Long Experience Report," in *Proceedings of Technical Symposium on Computer Science Education*, 2023, pp. 507–513.

[25] J. L. Bishop and M. A. Verleger, "The flipped classroom: A survey of the research," *Proceedings of ASEE National Conference*, pp. 23–1200, 2013.

[26] J. O'Flaherty and C. Phillips, "The use of flipped classrooms in higher education: A scoping review," *The Internet and Higher Education*, vol. 25, pp. 85–95, 2015.

[27] J. R. Anthony Robins and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 29, no. 3, pp. 221–263, 2019.

[28] M. Guzdial, "Why i use media computation for teaching computer science," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 2014, pp. 89–94.

[29] Y.-C. Tu, V. Terragni, E. Tempero, A. Shakil, A. Meads, N. Giacaman, A. Fowler, and K. Blincoe, "Github in the classroom: Lessons learnt," in *Proceedings of the Australasian Computing Education Conference*, 2022.

[30] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2005.

[31] D. Janzen and H. Saiedian, "Test-driven learning in early programming courses," in *Proceedings of the 39th International Conference on Frontiers in Education*, 2005, pp. T2G–1–T2G–6.

[32] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2000.

[33] B. S. Clegg, P. McMinn, and G. Fraser, "An empirical study to determine if mutants can effectively simulate students' programming mistakes to increase tutors' confidence in autograding," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 2021, pp. 1055–1061.

[34] Y. Jia and M. Harman, "Analysis and comparison of mutation testing strategies," *Software Testing, Verification & Reliability*, vol. 20, no. 3, pp. 239–262, 2010.

[35] D. Schuler and A. Zeller, "Evaluating the effectiveness of test suites in detecting faults using mutation testing," *Software Testing, Verification & Reliability*, vol. 23, no. 5, pp. 366–394, 2013.

[36] M. Gouveia and A. Costa, "Black-box testing in software engineering education: A case study," *IEEE Transactions on Education*, vol. 66, no. 1, pp. 45–58, 2023.

[37] K. Maurer and D. Wong, "Promoting design flexibility through black-box testing," *Journal of Software Testing and Verification*, vol. 30, no. 4, pp. 372–389, 2022.

[38] M. Aniche, *Java code metrics calculator (CK)*, 2015, available in https://github.com/mauricioaniche/ck/.

[39] P. Denny, V. Kumar, and N. Giacaman, "Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education*, 2023, p. 1136–1142.

[40] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001, pp. 92–95.

[41] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, p. 170, 2022.

[42] M. P. Robillard, M. Nassif, and M. Sohail, "Understanding test convention consistency as a dimension of test quality," *ACM Transactions on Software Engineering and Methodology*, 2023.

[43] GitHub, "Github copilot now available for teachers," 2023, accessed: 2024-11-01. [Online]. Available: https://github.blog/news-insights/product-news/github-copilot-now-available-for-teachers

[44] Q. Fu, Y. Zheng, M. Zhang, L. Zheng, J. Zhou, and B. Xie, "Effects of different feedback strategies on academic achievements, learning motivations, and self-efficacy for novice programmers," *Educational technology research and development*, vol. 71, no. 3, pp. 1013–1032, 2023.

[45] "Selenium: Web application testing framework," 2024, https://www.selenium.dev.
[46] "Cypress: End-to-end testing framework," 2024, https://www.cypress.io.
[47] "Autoit: Automation tool for gui applications," 2022, https://www.autoitscript.com.
[48] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using gui screenshots for search and automation," in *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, 2009, pp. 183–192.
[49] D. E. Libes, "Expect: A power tool for systems administration automation," *NIST Publications*, 1995. [Online]. Available: https://www.nist.gov/publications/expect-power-tool-systems-administration-automation
[50] R. Izuta, S. Matsumoto, H. Igaki, S. Saiki, N. Fukuyasu, and S. Kusumoto, "Detecting functional differences using automatic test generation for automated assessment in programming education," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 526–530.