# A Direct Manipulation Programming Environment for Teaching Introductory and Advanced Software Testing

Maximilian Georg Barth*
bmaximilian@ethz.ch
ETH Zürich
Zürich, Switzerland

Sverrir Thorgeirsson*
sverrir.thorgeirsson@inf.ethz.ch
ETH Zürich
Zürich, Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zürich
Zürich, Switzerland

## ABSTRACT

Despite software testing being an important part of computing education, the subject has not been strongly emphasized in the tertiary-level computing curriculum. This has been attributed to the difficulties involved in teaching the subject, for example because existing tools are difficult to navigate and require students to learn additional programming syntax in order to use them. In this paper, we introduce the new visual programming environment TestVision for learning unit testing and mutation testing in a hands-on, constructivist setting where test cases can be composed and executed via direct manipulation of visual elements rather than by writing code. We conducted a twofold study on the system with tertiary-level students. In the first study, twenty tertiary-level students were asked to complete interactive tutorials within the environment while their screens were recorded. We used the data to develop a taxonomy of the usability errors in the system. In the second study, nine graduate students from a course on software testing were invited to test the system while receiving a lecture on how to use it, after which their feedback was collected via free-form survey responses and the User Experience Questionnaire Plus. According to the results, students perceive the environment as well-designed, easy to use, and a useful tool for learning software testing.

## CCS CONCEPTS

• **Human-centered computing → Visualization systems and tools**; • **Social and professional topics → Computing education**.

## KEYWORDS

software testing, direct manipulation, programming by demonstration, tertiary education, mutation testing

## 1 INTRODUCTION

Software testing education is important for several reasons. First, as highlighted in a recent, comprehensive metareview on the subject [17], computer code has become increasingly complex and widespread across almost every industry, which has caused a growing need for quality assurance analysts with the relevant training. In fact, besides being just the purview of specialists, testing has also come to be considered "one of the most important skills an engineer should have" [1] and the ability to use processes such as unit testing has been identified as one attribute that distinguishes great software engineers [22], underscoring the need to incorporate the subject in general computing education. Additionally, besides its role in training software developers, a second, less instrumentalist argument for software testing education is how it can contribute towards critical thinking and problem-solving skill development; for example, testing and debugging have been identified as important computational thinking practices [11].

In spite of this, we believe that testing has received relatively little attention from the computing education community in comparison to, for example, code composition and comprehension. This may be attributed to the perception that the subject is difficult to teach [1]. In particular, one challenge is the lack of suitable testing tools; Garousi et al. [16] find that "to do (automated) testing, students need to learn new tools and libraries, which are often not easy to learn, especially in the earlier (first or second) years of a degree." This view is well-supported by their review of the literature; for instance, Clarke et al. [5] claim that students "continue to be frustrated" over difficulties in finding suitable testing tools. Developing suitable software is not simple; for example, it has been suggested that such software is particularly hard for beginners since learning additional syntax for defining tests compounds the difficulty of learning programming syntax in general, which is hard enough as it is [8]. More generally, Neto et al. [25] find that due to the difficulties involved in having students "deal with peculiarities of the specific techniques and tools for software testing," the topic may not be given the attention it deserves, which can contribute to the feeling among students that testing is hard or unimportant.

In this systems and tools paper, we aim to support students and educators by introducing a new programming environment for teaching software testing called *TestVision*. For one, we wish to address the aforementioned drawbacks in other tools by enabling students to define and execute test cases visually, as opposed to requiring students to master yet another domain-specific language with its unique syntax. However, we only see this as a necessary first step of our implementation. More generally, our vision is based on the idea that students struggle with testing for similar reasons as they struggle with programming or with using creative, interactive systems in general, namely what Don Norman identifies as the "Gulfs of Evaluation and Execution" [26], meaning that there's a cognitive gap between users' intentions and the system's provisions, as well as between the system's responses and users' interpretations. In the context of testing, this implies that students do not see immediate and clear feedback linking the tests they write to the outcomes those tests produce. To address this, we believe that our system should (i) keep the program state visible at all times, thus allowing for live, iterative testing where the effects of different inputs and code changes can be observed instantly, and (ii) allow programmers to work with the visualization directly using an interaction paradigm called direct manipulation.

*Maximilian Georg Barth and Sverrir Thorgeirsson are co-primary authors.

To achieve this vision, we have built TestVision on top of Algot [34, 37], a visual, graph-based programming language that supports liveness and direct manipulation and maintains a constant visual representation of the program state. We believe that Algot should be a suitable choice since controlled, experimental studies on secondary and tertiary-level students have found that the language can be learned very fast, both for composing code [33, 35, 36] and comprehending it [19], and does not induce high cognitive load in the domains that have been tested. Besides supporting users in defining test cases visually, we allow students to assess the quality of their test cases via visual branch coverage metrics and mutation testing, which allows students to define and test mutants in the same way that they can define unit tests. Our goal is to both support secondary-level students by making software testing more accessible and immediate through a visual and interactive tool and to support tertiary-level students by providing advanced test quality assessment features such as mutation testing and branch coverage metrics.

To our knowledge, applying visualization and direct manipulation in the context of software testing education is a novel contribution. Using our system, we aim to contribute to all three key research questions identified by the Garousi et al. review [17]; by making testing more accessible and immediate, we hope to (1) change the mental models of students to appreciate software testing and (2) motivate them to engage with the topic, and with our test quality modules, we hope to (3) introduce a measure that "help[s] educators assess the quality of software testing beyond 'just' code coverage."

In the remaining parts of the paper, we will describe the context and background from which the system arises (Sections 2 and 3) and offer a comprehensive overview of how it works (Section 4). We will also describe the methodology and design process behind the two studies in which we tested it (Section 5) and report on the usability taxonomy and other results we gathered from our studies (Section 6). Finally, we will discuss our results and the implications of our work (Section 7). We note that the system itself can be accessed at this link: https://testvision.algot.org.

## 2 RELATED WORK

In their 2020 review, Garousi et al. [17] identify 62 research papers that propose or discuss a specific tool for software testing education. Approximately a third of these papers (20) discuss three tools; (i) Web-Cat [31] (from 2003), an early web environment with features that were new at the time (such as auto-grading and online submissions of assignments), (ii) WReSTT-Cyle [6] (from 2010; now called STEM-CYLE), a collection of online tutorials on testing with support for social networking and gamification, (iii) Code Defenders [30] (from 2016), a novel multiplayer game for teaching mutation testing, with one objective of "[engaging] learners in mutation testing activities in a fun way." These approaches all involve textual programming.

We are aware of few systems using visual approaches to software testing and none that involve direct manipulation. Of the former, two early tools for testing education incorporate visualization to some degree; Light Views [29] from 2000, which was "one of the first environments used for software testing education" [38], could be used to visualize solution paths and state changes for programs written in Java, helping students generate comprehensive test cases. An unnamed educational system from 2005 would also provide a "visual representation of the testing mechanism" according to its authors [9]. The available information on these systems appears to be limited.

From the adjacent area of software built for testing visual programming languages, Whisker [18, 32] is a tool for automated and property-based testing of programs written in the visual block-based language Scratch. Although syntax errors in Scratch are impossible due to the block-based language paradigm, other bugs will still occur [12, 15], so a system like Whisker can be useful; an evaluation on a large number of Scratch programs found that Whisker can achieve high error coverage under a fully automated approach [12]. However, it does not appear that Whisker was designed with software testing education in mind in particular, but rather to help Scratch programmers write more correct programs.

## 3 BACKGROUND

Our system, TestVision, builds on top of the visual programming language Algot and adds features that allow users to test their programs and assess the quality of their tests. This is achieved through coverage metrics and mutation testing. The goal of the system is to introduce users to tools that can be used to improve the quality of a test suite and help them write better tests, a skill that many software engineers lack [4]. The following subsections will cover the theory behind the features implemented in our system.

### 3.1 Algot

Algot is a visual programming language designed to bridge the syntax-semantics gap through direct manipulation of the program state, which is represented as a graph that is always visible to the programmer [34, 37]. Each node of the state graph contains a value of type integer, float, or string. By applying built-in or user-defined operations to nodes in the graph, the program state can be modified, which will result in immediate changes to the program state. For example, operations can introduce new nodes to the graph, delete existing ones, introduce new edges between nodes, or change the value stored in a node. It is also possible to conditionally apply operations through queries, which are special built-in operations that can be used to check the conditions within the graph state.

The central part of Algot's interface is the playground, which contains the current representation of the program state. When executing operations on nodes of the graph, the values are instantly updated, allowing the user to directly interact with the results of the operation. For example, if the playground contains three nodes with the values 1, 2, and 0, selecting the "Sum" operation and clicking on the nodes in the listed order will store the result of adding 1 and 2 in the node that previously contained the value 0. This makes it very easy for users to test different operations and experiment in Algot.

Algot also allows users to create their own operations without having to write any code. This is done through the programming-by-demonstration paradigm, where users apply operations sequentially on input nodes they define. This process occurs in an environment similar to the playground, making it easy for users to create new operations. It is also possible to conditionally execute steps of an operation using queries. While Algot does not support loops, it allows users to recursively call an operation. For more details on the system, we refer to a 2022 overview [37].

Several controlled, experimental studies on Algot have been conducted. Two studies found students programming in Algot experienced lower cognitive load when creating programs in Algot compared to Python for the same tasks, one making use of an electroencephalogram [35, 36]. A study on secondary school students found that students learning recursion in Algot performed better on a recursion-related posttest than students learning about recursion using the block-based language Scratch [33]. Last, a study on program comprehension in Algot found that students understood programs better when presented in Algot than in Python [19].

### 3.2 Unit Testing

In unit testing, the program is divided into individual components, which are tested separately. These components are usually the smallest testable parts of a program, such as functions or methods. In Algot, this would correspond to a user-defined operation. The goal is to test each unit independently to ensure that it performs as expected, rather than testing the system as a whole. Unit testing is commonly used in practice and almost every programming language has its own unit testing framework [10].

To determine how much of the code is covered by a test suite and thereby assess its thoroughness, metrics such as branch coverage are used. With branch coverage, the developer can check that every possible branch in the program is covered by at least one test, ensuring that every line of code is tested at least once. In C-like languages, branches are introduced by if, switch, and ternary statements, as well as loops. Although branch coverage is widely used by real-world developers, it is inadequate for assessing the fault detection capability of a test suite because it only checks whether each branch is executed, not whether the state of the program has been affected by a bug [28].

### 3.3 Mutation Testing

Mutation testing is a software testing technique for assessing the quality of a test suite. It introduces faults into a program using mutation operators, which emulate common programming mistakes, such as switching a logical OR operation for a logical AND. The core assumption is that if a test suite can detect small changes in a program, it can also detect more complex faults. This is called the "Coupling Effect" [13]. All generated mutants are then run against the test suite to see whether the existing tests catch all introduced mistakes. If a mutant fails at least one test, it is considered killed. If a mutant manages to pass all tests, the test suite should be expanded to kill it.

Mutants that are killed by almost every test are considered trivial, whereas mutants that pass almost every test are considered stubborn. It is also possible for mutants to be equivalent to the original program. In this case, it is impossible to kill them. Detecting equivalent mutants requires human interaction since the problem of determining whether two programs are equivalent is undecidable.

A commonly used metric in mutation testing is the mutation score, which is calculated by dividing the number of killed mutants by the total number of mutants. The goal of the developer should be to achieve a high mutation score to ensure that the test suite is comprehensive enough. The mutation score is considered the state-of-the-art coverage metric due to its ability to measure the fault detection capabilities of a test suite [28].

Since mutation testing can be computationally expensive due to the generation of a large amount of mutants and the need to run each mutant against the entire test suite, the concept of weak mutation testing exists. In weak mutation testing, the program state is compared immediately after executing the mutated statement, instead of comparing the final program output. This usually results in a decreased runtime; however, it can lead to mutants passing tests they would have failed in strong mutation testing.

Our web search on the topic showed that mutation testing is taught in software testing courses at many universities around the world. It has been shown to be an effective tool for teaching software testing [3] and can be used by students to assess the quality of their test suites.

## 4 SYSTEM

Algot is implemented as a web application using TypeScript with the React-based Next.js framework and MongoDB for persistent storage. TestVision builds on top of Algot, adding new testing capabilities aimed at familiarizing beginners with the basics of software testing. Additionally, it includes mutation testing features and branch coverage metrics, which can be used to assess the quality of test suites. These features are targeted towards programmers who are already familiar with the basics of testing and are intended for use in software testing courses. The goal is to not only teach students how to write tests but also how to write good tests and judge their adequacy.[1]

We argue that the visual and interactive environment of Algot can deepen students' understanding of software testing by allowing for experimentation and exploration. This knowledge can then be applied to other programming languages, hopefully enabling them to write better tests - an important skill for any software engineer.

The new testing features include:

(1) **Testing Tab**: Located in the left side-panel, this tab allows users to create and run tests.
(2) **Test Editor**: A dedicated interface for writing tests, based on the Algot operation editor.
(3) **Branch Coverage Metrics**: Display the extent of code execution and help assess the effectiveness of the tests.
(4) **Step-Through Debugger**: A tool for examining test execution step by step.
(5) **Mutation Testing Screen**: This interface enables the automatic generation of mutants and running them against test suites. This allows the user to check the quality of a test suite and improve their test-writing skills.
(6) **Tutorials**: Comprehensive guides for both unit testing and mutation testing.

The system's features were designed with Algot's programming-by-demonstration paradigm in mind: for example, tests are defined by visually constructing the expected inputs and outputs, similar to how operations are defined, and do not require the user to write

[1] We note again that the system itself can be accessed at this link: https://testvision.algot.org.

any code or assertions. The mutation testing features allow the user to experiment with different mutation operators and receive instant visual feedback. This enables users to interactively learn about more advanced concepts of mutation testing, such as weak mutation testing. All new features will be discussed in detail in the following subsections. Note that we have also included in our supplementary materials three CS1-style program examples that show how TestVision works.

## 4.1 Testing Tab

All new testing-related features introduced by TestVision can be accessed via the testing tab in the left side-panel of Algot, as shown in the top left of Figure 1. This extension of the existing interface allows users to efficiently switch between operations and their corresponding tests. All tests are grouped by operation into individual test suites, following the convention in software engineering of testing each component separately. The goal is to implicitly teach this practice to users of TestVision.

At the top of the testing tab, the current passing rate of the selected test suite and the achieved branch coverage are displayed. These metrics provide users with a quick estimation of how effective their tests are and introduce them to commonly used metrics in software testing. Also at the top of the sidebar are controls to create new tests, run all tests, and open the mutation testing screen for the current test suite. The mutation testing screen will be covered in detail in a later subsection.

Below the controls, a list of all tests for the currently selected operation is displayed. Each entry in the list contains the name of the test and controls to run, debug, or delete it. By clicking on the test, the user can open the test editor. It is also possible to rename the test by double-clicking its name. This behavior is consistent with the existing operation tab in Algot and allows users to assign more meaningful names to their tests. Below the test name, additional information about the test is displayed, including the achieved branch coverage of the individual test and whether it has passed or failed, along with possible error messages. This information is automatically displayed when the test is run and can be hidden by the user.

## 4.2 Calculating the Branch Coverage

Metrics are an important tool in software testing, allowing developers to judge how much of the code they are covering with their tests and assess the effectiveness of their test suite. We believe it is important to introduce beginners to these metrics, as they are very helpful for writing good tests. For this reason, Algot's interpreter was extended to allow for the calculation of the branch coverage of an operation execution. An example of this can be seen in the left sidebar at the top left of Figure 1.

Branches in Algot can only be introduced by queries, which evaluate to either true or false. Each query represents a decision point in the execution path, introducing branches in the process. Actions taken during an operation can be conditional on the result of one or more queries. However, queries cannot be dependent on the result of other queries. In the case where an action is dependent on multiple queries, the conjunction of the query results is taken, only evaluating to true if all subqueries evaluate to true.

The branch coverage achieved by a test is calculated by aggregating the results of queries for each action dependent on one, across recursive calls. For example, if a query evaluates to true during the first call and to false in the second recursive call, the aggregated result for that query would contain both true and false. Conversely, if a query only results in false across all calls, the aggregated result would contain only false. The number of covered branches is then determined from these aggregated results. Once the operation terminates, the total number of branches is calculated by doubling the number of actions dependent on queries, since each query can result in either true or false. If an action depends on the result of multiple queries, their conjunction is treated as a single branching point.

Consider the following operation as an example (shown in Figure 2): Take two nodes as input arguments and decrement the value of the first node until it is equal to the value of the second node or 0. This operation has two actions, both dependent on the same two queries: "Is the value of the first node not equal to the value of the second node?" and "Is the value of the first node not equal to zero?".

If we call this operation with two nodes containing the values 4 and 2, we will have two calls—one initial call and one recursive call. In the first iteration, both queries evaluate to true, decrementing the value of the first node and triggering the recursive call. In the second recursive call, the value of the first node is again decremented, and the operation terminates, as both nodes now have the same value and the conjunction of the queries evaluates to false.

We can now determine the total number of branches by counting the number of queries performed during the operation. Since we had two calls and each call evaluated two queries, we end up with 4 branches. As the query for the first action only resulted in true, and the query for the second action evaluated to both true and false across the recursive calls, we can conclude that 3 out of the 4 possible branches were covered. This results in a branch coverage of $\frac{3}{4} = 75\%$.

## 4.3 Test Editor

Tests in TestVision consist of three parts: the input graph, the input nodes, and the output graph. The input graph defines the structure of the test input, and the input nodes are the nodes within this graph that are passed to the operation as arguments. An example of this can be seen at the top left of Figure 1. The output graph describes the expected shape and values of the output.

To determine whether a test has passed, the operation is executed on the input nodes, and the output is compared to the output graph defined in the test. If they match, the test passes; if they differ, the test fails. This approach is based on traditional testing methods used in textual programming languages, allowing users to apply the concepts they learn in TestVision to other programming languages. However, it requires solving the graph isomorphism problem to determine the test outcome. Although the graph isomorphism problem is not known to be solvable in polynomial time [14], this has not been an issue in the user studies and testing. There exists a quasi-polynomial algorithm [2], but it has not been implemented in TestVision as it has not proven necessary.

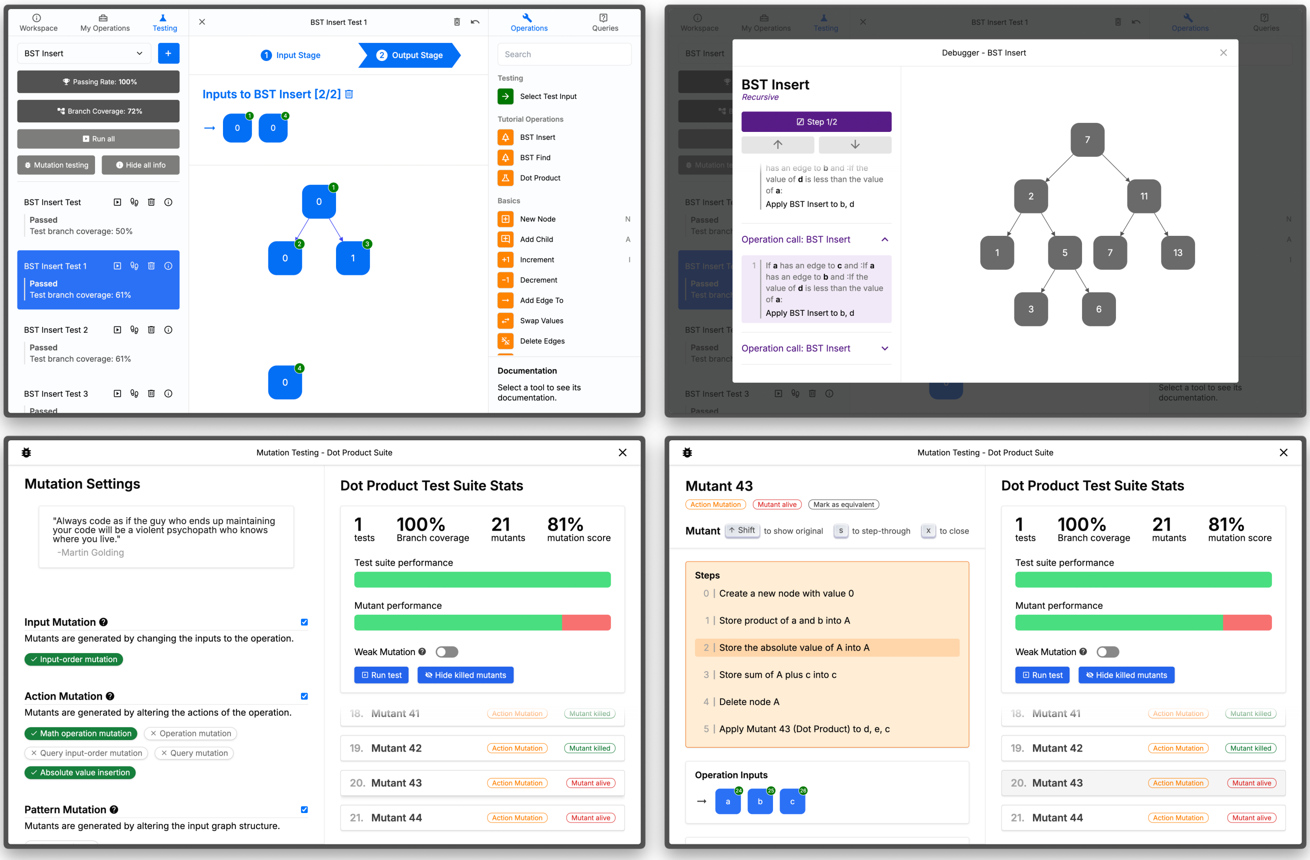


**Figure 1: Different components of Algot's new testing features, clockwise from the top-left: (a) First stage of the test editor with the new testing tab on the left-hand side (b) Step-through debugger (c) Mutation inspection screen of an action mutant (d) Mutation testing screen with the three different mutation operation types on the left-hand side**

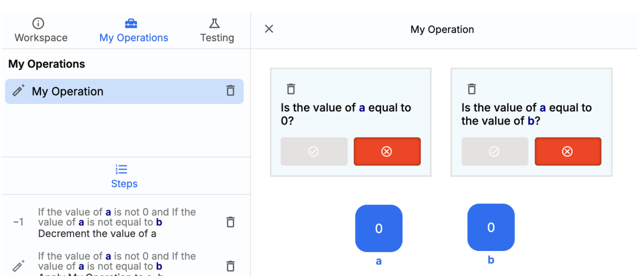


**Figure 2: An example operation for calculating branch coverage in TestVision (screenshot edited to reduce whitespace).**

Therefore, the editor must allow the user to define all three parts of a test: the input graph, the input nodes, and the output graph. It is based on the existing operation editor of Algot and is separated into two stages. By reusing parts of the existing interface, users can apply what they have already learned about writing operations in Algot to the new testing features. In traditional text-based languages, programmers often have to learn a separate testing framework and write boiler-plate testing code in which they can insert their assertions. This is because, in many languages, testing is not part of the standard library and third-party solutions have to be important.

For beginners, this can be a complicated process, especially because they have to learn language-specific testing methods. In TestVision, this process is streamlined, allowing the user to concentrate on the concepts of testing itself rather than learning a testing framework. We believe that this improves the learning process for beginners, enabling them to focus on the ideas and thinking patterns of testing, rather than memorizing how to use a testing framework. First, the user defines the input graph and selects the input nodes. This is done in the same way as any other graph is defined in Algot. In this stage, a new tool is added to the right sidebar, allowing the selection of the input arguments to the operation and their order. Not all nodes of the graph have to be selected as inputs, since Algot operations can also operate on nodes that are connected to input arguments.

In the second stage, the user defines the expected output graph. This is also done in the familiar graph editor.

## 4.4 Step-through Debugger

The step-through debugger can be opened for any test and used to inspect why a test is failing, as shown at the top right of Figure 1. This follows Algot's philosophy of always displaying the program state and allows users to see the results of each individual step, which may help them better understand how tests are run and evaluated. To enable the debugger, Algot's interpreter was extended to allow for partial function applications. This adjustment allows each action performed during an operation to be applied individually, enabling users to step through functions.

At the center of the debugger is a non-editable graph view of the current program state. Next to the graph view is a list of all steps executed during the operation, grouped by operation call. These groups can be collapsed, making it easier to quickly navigate through recursive functions. The current step is highlighted in the list, and the user can continue stepping through the operation by using the arrow keys. With each step, the state of the graph is updated, allowing the user to see the effects of each action performed by the operation.

## 4.5 Mutation Testing Screen

The mutation testing screen can be opened for any test suite and allows for automatic mutant generation and running the mutants against the test suite. An example of this can be seen at the bottom left of Figure 1. The goal of mutation testing is to give users an additional tool, beyond branch coverage, to help them assess how effective their test suite is. Mutation testing has been proven to be effective for teaching software testing, as it gamifies the testing process [3].

Operations in Algot involve a set of actions performed on a set of input nodes that are part of a graph. These three aspects can be mutated and constitute the three different mutant generation categories:

(1) **Input Mutation**: This category mutates the input arguments to the operation. There is one mutation operator, which changes the order of the arguments to the operation.
(2) **Action Mutation**: This category mutates the actions executed by an operation. There are five different mutation operators available, all based on common mutation operators from textual languages, such as switching mathematical operators [20].
(3) **Pattern Mutation**: This category mutates the input graph. It is equivalent to swapping references within a struct in a C-like language. There is one mutation operator in this category, which swaps the children of a node if it has more than one child.

Every mutation operator can be enabled individually, automatically updating the list of all mutants on the right-hand side of the screen. When hovering over the button of a mutation operator, a popup appears explaining what it changes in the operation. This is illustrated at the bottom right of Figure 1, where the action with index 2 was mutated. There are also tutorial screens for each mutation category, showing visual examples of how they work, which can be opened by clicking on the help button next to the category name. Each entry in the list of generated mutants is labeled by category and can be inspected by clicking on it. Using the arrow keys, users can quickly navigate through all generated mutants and inspect them. These features allow users to experiment with different mutation operators very easily, while also providing instant visual feedback. We believe that the quick feedback loop can be very beneficial for the learning process.

The mutant inspection screen consists of three parts, corresponding to each component of an Algot operation: the actions executed by the operation, the input nodes to the operation, and the input graph. The part changed by the mutant is highlighted in orange, making it easy to spot the applied mutation. For mutations in the action mutation category, the exact step that was altered is additionally highlighted. By pressing the Shift-key, users can toggle the view between the original operation and the mutant. This feature helps users understand how the applied mutation operator works and how it changes the operation. If users have trouble understanding why a mutant is not killed by the test suite, they can make use of the step-through debugger to see why it passes a specific test case.

At the top of the mutant inspection screen, there are labels showing the current state of the mutant. After running the tests on the mutants, the labels are updated to indicate whether the mutant is alive or killed. By hovering over the label, users can see which tests the mutant passes or fails. If the mutant is killed by 90% or more of the tests, it is considered trivial and a badge is displayed next to its name. If it is only killed by 10% or fewer of the tests, it is considered stubborn, and a badge is shown. Users interested in learning more about trivial and stubborn mutants can hover over the badges, rewarding their exploration.

There is a button that allows users to mark a mutant as equivalent, as detecting whether a mutant is equivalent requires human intelligence [7], with an explanation available on hover. When a mutant is marked as equivalent, it is excluded from the calculation of the mutation score. The mutation score, a common metric in mutation testing [20], is calculated by dividing the number of killed mutants by the total number of mutants and is used to judge the effectiveness of a test suite.

Above the list of all mutants is a box displaying metrics about the mutants, including the number of generated mutants, the aforementioned mutation score, and the achieved branch coverage of the test suite. These statistics are also visualized using progress bars, with red and green colors indicating the performance of the test suite. For example, the use of green helps users intuitively understand that a high mutation score is desirable. The box also contains controls to run the mutation test and to hide killed mutants. Since it is easy to generate a large number of mutants, this feature helps users identify mutants that are not killed by the test suite.

Additionally, there is a switch that allows users to toggle between strong and weak mutations. With strong mutation, the tests are run in their entirety, checking whether the output matches the expected test output. However, since mutation testing can generate a large number of mutants and each mutant is run against every test, this can lead to long runtimes. To address this, weak mutation testing only executes the test until after the mutated action is performed, resulting in faster runtimes. Weak mutation testing is commonly used in practical applications of mutation testing [27], but its results are less accurate since the states are compared early, which may lead to mutants being mislabeled as alive or killed.

### 4.6 Tutorials

Algot has a tutorial that teaches users how to create and use operations, guiding them through the interface. This tutorial is presented in a floating window that provides instructions and positive visual feedback upon task completion. For TestVision, two additional tutorials have been added.

The first tutorial guides users through the new testing tab, test editor, and step-through debugger. It starts by introducing a new operation that calculates the greatest common divisor (GCD) of two nodes. Users are instructed to apply it to nodes in the playground to understand its usage. Next, they create a simple test for the GCD operation and run it, which includes an overview of the new test editor. After running the test, the concept of branch coverage is introduced, explaining how it helps assess the effectiveness of a test suite. Users then test how the GCD operation handles divide-by-zero exceptions, learning about the importance of testing edge cases. Finally, they use the step-through-debugger to identify the bug causing the mishandling of zero inputs.

The second tutorial introduces the mutation testing features of TestVision and covers the mutation testing screen. It begins by adding an operation that computes the dot product of two vectors. Users are asked to test this operation in the playground, similar to the previous tutorial. Next, users write a test for the operation, covering a specific input case provided by the tutorial. The tutorial then explains mutation testing, how it works, and how it can be used to assess the quality of a test suite. Users are instructed to test different mutation operators and to write new tests for mutants not killed by the test suite. This also includes an introduction to the mutant inspection screen and the mutation score. After killing all mutants but one, users are asked to inspect why the remaining mutant is not killed by the test suite, learning about equivalent mutants and how to mark them accordingly in the interface.

The general philosophy behind the tutorials is to avoid overloading users with information. This is achieved by gradually introducing users to the concepts of testing and mutation testing, and allowing them to learn more details through exploration. For example, users can read hover texts and open information screens, which rewards interested users. The tutorials aim to teach users how to write tests in TestVision and how to use metrics such as branch coverage and the mutation score to assess the quality of their tests. We argue that these skills can be applied to other programming languages, enabling users to evaluate the quality of their tests and improve their test-writing capabilities.

## 5 METHODS

After receiving ethics approval from our institution, we sought to evaluate the usability of the software in two separate user studies:

- First, with the help of the Decision Science Laboratory at ETH Zürich, we recruited twenty participants from a large volunteer pool of tertiary-level students (>10,000). The precondition for participating was to have completed at least one course in computer programming, but they were not required to be enrolled in a computer science degree program. We asked the students to proceed through an interactive, digital tutorial on the system in a computer laboratory while their screens were recorded. To help us find how usable the system is, we sought to (i) determine how many participants could successfully complete the tutorial, (ii) identify and classify design errors encountered during the session, and (iii) analyze the feedback collected from their participants, including textual feedback and their responses to a 10-stage Likert scale on whether they believe the system can help them learn about software testing.
- Second, we recruited nine participants from a course on automatic software testing for master's students in computer science. All registered students in the course were invited to participate. First, before the tutorial began, the students were asked on a free-form survey question to identify which topics from the course they found challenging. Then we presented the system live while the students followed our presentation on their own laptops and tested the system for themselves. Afterwards, we asked the participants to rate their perceptions of the system using the standardized User Experience Questionnaire Plus [23] with the four scales *novelty, usefulness, intuitive use* and *clarity*. Each scale is measured by taking the average of a 7-point rating of four adjective-antonym pairs. For example, to assess the *usefulness* of the system, students will consider the pairs *useless–useful, not helpful–helpful, not beneficial–beneficial, not rewarding–rewarding*.
  We also asked participants to offer their feedback using five free-form survey questions:
  - (Q1) Please describe your impression of the software.
  - (Q2) Could you describe any challenges you can identify with the software?
  - (Q3) Do you think that the software can help you with your learning goals?
  - (Q4) What improvements or additional features could improve the software?
  - (Q5) Do you have any other comments regarding the software that you would like to share?

Participants were also asked about their age and gender using free-form questions. The participants in the first study were also asked about their education level and their familiarity with software testing using a slider-style question.

Note that we anticipated that we would make substantial improvements to the system in response to the feedback collected from the participants in the first study. Therefore, we did not anticipate that the systems shown to the participants in the two separate user studies would be precisely the same.

## 6 RESULTS

### 6.1 First session

Twenty participants attended our first session, all of which completed the study. Ten participants were female and ten were male. Of the nineteen participants who answered our question about their age, their median age was 23 (range: 21 to 31). Thirteen participants were enrolled in a master's program and seven were enrolled in a bachelor's program. None had used Algot before. All students had completed at least one course in computer programming. Seventeen chose to answer our question about their familiarity with software testing; according to the slider we used, the average value was 19%, indicating low familiarity.

*6.1.1 Survey results.* We also analyzed free-form responses to the question “Do you have comments about anything that could be improved?”. Thirteen students chose to respond. Four students mentioned the interactive tutorial, for example “I got stuck on one explanation of the tutorial. It would be nice if it is possible to still continue with the tutorial.” Another suggested that the tutorial should have a “back” button. One student said “I was first a bit overwhelmed by the amount of bars and tabs that [I] can open.” One student asked if it was possible to use non-directed edges and another proposed “being able to add steps in between steps and being able to rearrange steps with clicking and dragging.” Other comments mentioned various errors ranging from typographical errors to more technical ones, e.g., “[o]pening the GCD operation did only show me node a and the isZero query for b, the other nodes (b,A) were missing, though I somehow got Algot to show me the other nodes once, but then the operation [didn’t] work anymore.” The same student mentioned that “the overall UI, UX, how to do stuff in Algot” was clear.

All twenty participants answered the question on how much the system could help with understanding software testing. The average response was 5.2 ($\sigma$ = 2.0, min = 3, max = 9) on the 10-point scale used.

*6.1.2 Usability Taxonomy.* We analyzed all twenty screen recordings taken during the first user study and identified and classified usability problems of the new testing features and interfaces introduced by TestVision. Since the first session targeted individuals with no prior testing knowledge, the mutation testing parts of the software were not included in this study.

The classification of the identified usability problems was performed using the Usability Problem Taxonomy (UPT) by Keenan et al. This model assists in identifying common issues in user interfaces and has been shown to be reliable [21]. The UPT categorizes usability problems from both task and artifact perspectives. The task component addresses problems users encounter while performing tasks, whereas the artifact component focuses on difficulties related to objects within the user interface. Each component includes further subcategories, allowing for more specific classification, such as issues with object appearance, naming/labeling, and non-message feedback. If a usability problem cannot be further specified within a component, it is considered fully classified (FC). If there are additional subcategories available for classification, it is partially classified (PC). If it does not fit within the component’s categories, it is null classified (NC). After categorizing all usability problems, they can be grouped by their classification, facilitating the identification of common issues within the user interface.

From the UPT in Table 1, we can see that some usability issues are not specific to TestVision, namely points (1), (2), and (6). One interesting observation is that users were able to find a bug during the tutorial using the new testing features but were unable to fix this bug in the operation editor. It is difficult to determine whether this is due to the user interface, the students’ inexperience with Algot, or the difficulty of the task. Further usability studies would be valuable to explore this issue.

The second most common usability issue was that participants mistook the operation editor for the playground. Both use the same graph editor for different purposes: the playground displays the current state of the workspace, while the operation editor shows the implementation of a specific operation. If a user creates a new operation and does not have any nodes in the playground, the visual difference between the two is very subtle. This violates the principle of feedback in human-machine interaction, which states that users should receive information about what action has been performed [26]. This issue should be addressed by creating a more distinct visual separation between the playground and the operation editor, such as using a different background color.

Another non-testing related issue was that two users unintentionally created an infinite recursion while defining an operation by creating a new node and applying the operation itself on it. This behavior shows that users were unaware they were defining the operation, which led to the creation of hundreds of nodes, causing visible confusion. As with the previous issue, this problem can also be attributed to the lack of visual distinction between the playground and the operation editor.

Additionally, this confusion suggests a further problem, as users first click on the "New Operation" button before creating the recursion. This button looks very similar to the buttons for calling user-defined and built-in operations, even though it behaves very differently. This represents a lack of consistency in the user interface and could also explain why users were not aware that they opened the test editor, as users expect objects that look alike and are in close proximity to each other to behave similarly [26]. Although this was an issue for only two participants, it should be addressed by relocating the "New Operation" button or making it visually distinct from the operations below it.

The most common issue related to the new testing features was that participants were unsure what to set the output of a test to when it caused a divide-by-zero exception, a scenario covered by the testing tutorial. The task states: *If your test input results in undefined behavior, simply return the input graph unchanged (in the output stage of the test).* The issue could be due to the wording, which may need a clearer explanation. This might also be addressed by introducing error handling in Algot, making it more obvious what to do in the case of an exception.

A further issue discovered in the testing interface is that some users forgot to select the input nodes from the input graph in a test. This could be because this behavior differs from text-based programming languages that the participants were already familiar with, requiring them to define two types of inputs: the shape of the input graph and the actual nodes that are passed to the operation. If the user attempts to run a test without selecting the input nodes, an error message is displayed stating, *"Test failed: Wrong input count."* However, the user might be confused by this wording if they have already defined the input graph but forgot to select the arguments.

This issue could be addressed by improving the wording in the error message and explicitly prompting the user to select input nodes after defining the input graph in the test editor. Additionally, a new error message could be introduced when the user forgets to define an input graph, making it more distinct from the case where they simply forgot to select the input nodes. Another approach

| # | #Users | Problem Description | Artifact Classification | Artifact Outcome | Task Classification | Task Outcome |
|---|---|---|---|---|---|---|
| 1 | 8 | The user attempts to debug the GCD operation but does not succeed. | Artifact | NC | Task-facilitation | PC |
| 2 | 7 | The user confuses the operation editor for the playground. | Object appearance | FC | Task-mapping | PC |
| 3 | 5 | The user is not sure what to set the GCD output to in the case of an exception during the testing tutorial. | On-screen text | FC | Task-facilitation | PC |
| 4 | 4 | The user edits the GCD operation, making it impossible to complete the testing tutorial. | Direct manipulation | FC | Keeping the user task on track | FC |
| 5 | 3 | The user forgets to select the test input nodes when defining a test. | Non-message feedback | FC | Task-facilitation | PC |
| 6 | 2 | When creating a new operation, the user unintentionally applies the operation on itself. | Naming/labeling | FC | Keeping the user task on track | FC |

**Table 1: Usability Problem Taxonomy (UPT) [21] from the first user study with twenty participants, gathered from analyzing the screen recordings of the students.**

would be to move the selection of input nodes to a separate stage in the test editor, making it more difficult to miss.

Finally, there also was a usability issue where users accidentally edited the GCD operation added by the testing tutorial. In all cases, they removed the exit condition of the recursion, leading to error messages due to reaching the maximum recursion depth of Algot when running tests. The users believed they had found a bug in TestVision and stopped doing the tutorial, even though they had caused the exception themselves. In the study, all participants who started the testing tutorial but did not complete it stopped because they had edited the GCD operation. There are two possible fixes for this issue: either the GCD operation added by the testing tutorial should be made static so users cannot edit it, or a warning message should be displayed when the operation is edited. This issue might also be prevented by making the editor more visually distinct from the playground, as users may have been unaware that they were editing the operation, as discussed before.

## 6.2 Second session

Nine participants attended our second session, of which two were female and seven were male. The median age was 23 (range: 22 to 26). Before the session took place, many of the errors pointed out in the first session had been resolved. We note that anonymized results from this session can be found in our supplementary materials.

The normalized results from the UEQ+ can be seen in Table 2. The average value across the four scales chosen was 1.72, which is high according to a recent benchmark of the UEQ+ [24], with the score comparing favorably against the interfaces of well-known industry software. The scores had a relatively low standard deviation and were rather consistent; of the 130 rating points collected in total, only two (1.5%) were below 0. The free-form responses were well-aligned with UEQ+ results; all nine students left positive comments when asked for their impression (Q1), finding it "modern and clear [S1]" "good software with a clear UI [S2]," "it is a very fancy software. It's my first time to use it, yet I find it quite easy to start with [S4]," "it looks good and easy to navigate [S6]," "very well thought out and beautifully designed [S9]," "the software seems very helpful to solve/understand problems before coding them [S7]," "nice to see visually what happens during mutation testing [S8]" and "design very good, minimal and easy to understand [S5]." The least positive feedback came from student S8, who said "the software is usable, but there can be improvements such as when designing an operation, the close button "x" can be more obvious."

Eight of the nine students answered affirmatively that the software could help them with their learning goals (Q3), e.g., "Yes. The visual nature of the software helps a lot with that [S1]" and "I think it is very nice to learn about testing with it [S9]". Four students mentioned that it would be helpful for mutation testing in particular (e.g., "It helped me clear up on questions on mutation testing and solidify my course knowledge [S9]," "It is for sure easier to grasp mutation testing visually [S3]" and "Yes it can be useful to learn recursion for beginners and the feature about mutation testing is very well done and I would use it to actually understand better how mutation testing works [S1]"). The only ambivalent student wrote "Unsure. I'd have to really dig deep to try it myself [S8]."

| Scale | Mean | 95% CL interval | SD | Min | Max |
|---|---|---|---|---|---|
| Novelty | 1.89 | [1.21, 2.57] | 0.98 | 0.25 | 3.00 |
| Usefulness | 1.69 | [1.04, 2.34] | 0.87 | 0.25 | 2.50 |
| Intuitive Use | 1.56 | [0.64, 2.49] | 1.29 | -0.75 | 2.75 |
| Clarity | 1.78 | [1.21, 2.36] | 0.69 | 0.75 | 3.00 |

**Table 2: The normalized results from the User Experience Questionnaire Plus (UEQ+) with the mean, its confidence interval, standard deviation (SD), and minimum and maximum scores. Each rating represents the average response to four questions. The ratings are on a scale from -3 (most negative possible) to 3 (most positive possible).**

When probed for challenges with the software or improvements that could be made (Q2 and Q4), three students (S1, S6, and S9) mentioned that they would like to be able to assign new names to the nodes in the graph. S5, S7, and S8 suggested more documentation and/or additional guided tutorials. S2 proposed more features like support for object-oriented programming and floating points, and S9 mentioned support for additional testing concepts like symbolic execution. S3 said "the challenge I can identify is how to deploy the written program in Algot to real life." Seven of the nine students chose to leave additional comments about the software (Q5); two students praised the tutorial in particular ("really clear [S3]" and "'really good [S5]") but two students (S5 and S8) found it too fast or that it covered too much material. Two students (S2 and S4) praised the software implementation. The last remaining response to Q5 was a question on the software versioning.

According to the free-form question that was asked before the system was presented, students had identified the following course topics as challenging or difficult: symbolic execution (S1 and S9), mutation testing (S4 and S8), abstract interpretation (S6) and data-flow analysis (S1).

## 7 DISCUSSION

Our twofold study yielded two types of results; first, we captured operational data on how the system is used by its intended audience (tertiary-level students) and cast light on the usage challenges involved, and second, we captured perceptual data on how users experience the usability of the system. In addition to how the first dataset helped us improve the system, our hope is educators can use the usability taxonomy that we synthesized to (i) better understand the design and design history of the tool and (ii) be better prepared for how students might experience it when encountering it in the classroom. For example, our usability study helped us discover an unintended consequence of attempting to make the different system interfaces similar; our intention by having the playground and the testing module incorporate a similar design was to make it easier for students to apply their knowledge across different environments, but this also resulted in students having trouble distinguishing between them. To make the best use of the system, we believe that it would be effective to begin with direct instruction, teacher-led activities before students experiment with it on their own, similar to the setup we used in the second study.

Our perceptual data on the system usability was neutral in the first study session (average rating 5.2/10) but very high in the second study session (1.7/3.0 on the Usability Experience Plus). The data should be interpreted cautiously given the low number of participants (especially in the second study), but our qualitative interpretation of the free-form data also indicates that the system was very well-received and that direct manipulation can bring the subject matter closer to students. We believe that the difference in results can be attributed to two factors; first, the design of the system was improved somewhat between the two sessions, and second, the students in the latter session were all enrolled in a course on software testing, meaning that they were likely more interested in testing and more likely to understand how our system could be used.

With respect to the delimitations of the study and the threats to validity, we note that neither of our study sessions measured learning gains resulting from the system or related metrics such as task performance, cognitive load, or motivation. Instead, we measured students' perceptions on whether the system is usable and whether it can help them learn. Although the responses to the last question were positive, this does not mean that learning gains will necessarily materialize in practice. Furthermore, although the responses were collected anonymously, it is possible that student responses were impacted by acquiescence or social desirability bias. Additionally, although our study showed that most students in our first session could complete the interactive tutorials successfully within the time allotted, which we take as evidence for the system usability, we do not have evidence that they could also do so for more complex tasks.

We find that controlled studies comparing the system with traditional forms of teaching software testing would be useful. However, there are currently some challenges involved in conducting such a study due to the lack of a validated assessment instrument or a concept inventory on software testing, or more generally, a robust understanding of what competency with software testing looks like and can be measured. Therefore, in addition to encouraging others to explore our tool in their research, we also call for more research on validated measurements in the domain of software testing; of the 62 papers identified by Garousi et al. [17] on software testing education, none were on this subtopic.

## 8 CONCLUSION

Our paper presented TestVision, a system for teaching students about software testing within a direct manipulation and live programming environment that allows students to define tests without working with a single line of code. Our paper presents the design of the system, a twofold empirical study on it with tertiary-level students, and a usability taxonomy resulting from our study. We found that students had positive perceptions on the usability of the system according to the User Experience Questionnaire Plus [23] and free-form questions, and that they also believed that the system could help them learn.

## REFERENCES


[1] Maurício Aniche, Felienne Hermans, and Arie Van Deursen. 2019. Pragmatic software testing education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 414–420.

[2] László Babai. 2016. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing (STOC '16)*. ACM. https://doi.org/10.1145/2897518.2897542

[3] Martin Balfroid, Pierre Luycx, Benoît Vanderose, and Xavier Devroey. 2023. An Empirical Evaluation of Regular and Extreme Mutation Testing for Teaching Software Testing. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 405–412. https://doi.org/10.1109/ICSTW58534.2023.00074

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[5] Peter J Clarke, Andrew A Allen, Tariq M King, Edward L Jones, and Prathiba Natesan. 2010. Using a web-based repository to integrate testing tools into programming courses. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 193–200.

[6] Peter J Clarke, Debra L Davis, Raymond Chang-Lau, James Kiper, Yujian Fu, and Gursimran S Walia. 2017. Using WReSTT cyberlearning environment in the classroom. *124th American Society for Engineering Education (ASEE). ASEE* (2017).

[7] Benjamin S. Clegg, Jose Miguel Rojas, and Gordon Fraser. 2017. Teaching Software Testing Concepts Using a Mutation Testing Game. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. 33–36. https://doi.org/10.1109/ICSE-SEET.2017.1

[8] John Clements and David Janzen. 2010. Overcoming obstacles to test-driven learning on day one. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 448–453.

[9] Jim Collofello and Kalpana Vehathiri. 2005. An environment for training computer science students on software testing. In *Proceedings Frontiers in Education 35th Annual Conference*. IEEE, T3E–6.

[10] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. https://doi.org/10.1109/ISSRE.2014.11

[11] Dan Damelin, Lynn Stephens, and Namsoo Shin. 2019. Engaging in computational thinking through system modeling.@. *Concord* 23, 2 (2019), 4–6.

[12] Adina Deiner, Patric Feldmeier, Gordon Fraser, Sebastian Schweikl, and Wengran Wang. 2023. Automated test generation for Scratch programs. *Empirical Software Engineering* 28, 3 (2023), 79.

[13] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/c-m.1978.218136

[14] Scott Fortin. 1996. The Graph Isomorphism Problem. (1996). https://doi.org/10.7939/R3SX64C5K

[15] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common bugs in Scratch programs. In *Proceedings of the 2020 ACM conference on innovation and technology in computer science education*. 89–95.

[16] Vahid Garousi and Aditya Mathur. 2010. Current state of the software testing education in north american academia and some recommendations for the new educators. In *2010 23rd IEEE Conference on Software Engineering Education and Training*. IEEE, 89–96.

[17] Vahid Garousi, Austen Rainer, Per Lauvås Jr, and Andrea Arcuri. 2020. Software-testing education: A systematic literature mapping. *Journal of Systems and Software* 165 (2020), 110570.

[18] Katharina Götz, Patric Feldmeier, and Gordon Fraser. 2022. Model-based testing of scratch programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 411–421.

[19] Oliver Graf, Sverrir Thorgeirsson, and Zhendong Su. 2024. Assessing Live Programming for Program Comprehension. In *Proceedings of the 29th Innovation and Technology in Computer Science Education Conference (ITiCSE 2024)*. ACM, Milan, Italy.

[20] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. 2019. Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 114–124. https://doi.org/10.1109/ICST.2019.00021

[21] Susan L. Keenan, H. Rex Hartson, Dennis G. Kafura, and Robert S. Schulman. 1999. *Empirical Software Engineering* 4, 1 (1999), 71–104. https://doi.org/10.1023/a:1009855231530

[22] Paul Luo Li, Amy J Ko, and Jiamin Zhu. 2015. What makes a great software engineer?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 700–710.

[23] Schrepp Martin and Jorg Thomaschewski. 2019. Design and validation of a framework for the creation of user experience questionnaires. *IJIMAI* 5, 7 (2019), 88–95.

[24] Anna-Lena Meiners, Martin Schrepp, Andreas Hinderks, and Jörg Thomaschewski. 2023. A benchmark for the UEQ+ framework: construction of a simple tool to quickly interpret UEQ+ KPIs. (2023).

[25] Vicente Lustosa Neto, Roberta Coelho, Larissa Leite, Dalton S Guerrero, and Andrea P Mendonça. 2013. POPT: a problem-oriented programming and testing approach for novice students. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 1099–1108.

[26] Donald A. Norman. 1995. *THE PSYCHOPATHOLOGY OF EVERYDAY THINGS*. Elsevier, 5–21. https://doi.org/10.1016/b978-0-08-051574-8.50006-6

[27] A. J. Offutt and S. D. Lee. 1994. An Empirical Evaluation of Weak Mutation. *IEEE Trans. Softw. Eng.* 20, 5 (may 1994), 337–344. https://doi.org/10.1109/32.286422

[28] Ali Parsai and Serge Demeyer. 2020. Comparing mutation coverage against branch coverage in an industrial setting. *International Journal on Software Tools for Technology Transfer* 22, 4 (May 2020), 365–388. https://doi.org/10.1007/s10009-020-00567-y

[29] Sita Ramakrishnan. 2000. LIGHTVIEWS—Visual interactive Internet environment for learning OO software testing. In *Proceedings of the 22nd international conference on Software engineering*. 692–695.

[30] José Miguel Rojas and Gordon Fraser. 2016. Code defenders: a mutation testing game. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 162–167.

[31] Anuj Ramesh Shah. 2003. *Web-cat: A web-based center for automated testing*. Ph. D. Dissertation. Virginia Tech.

[32] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 165–175.

[33] Sverrir Thorgeirsson, Lennart Lais, Theo Weidmann, and Zhendong Su. 2024. Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon.

[34] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: An Educational Programming Language with Human-Intuitive Visual Syntax. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576166

[35] Sverrir Thorgeirsson, Theo Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. Comparing Cognitive Load Among Undergraduate Students Programming in Python and the Visual Language Algot. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon.

[36] Sverrir Thorgeirsson, Chengyu Zhang, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. An Electroencephalography Study on Cognitive Load in Visual and Textual Programming. In *Proceedings of the 2024 ACM Conference on International Computing Education Research (ICER '24)*. ACM, Melbourne, VIC, Australia.

[37] Theo B Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 80–94.

[38] Tamara Zivkovic, Drazen Draskovic, and Bosko Nikolic. 2023. Learning environments in software testing education: An overview. *Computer Applications in Engineering Education* 31, 6 (2023), 1497–1521.