Bachelor thesis

Computer Science and Engineering, 300 credits. Computer Engineer, 180 credits.

Java Auto Grader

Computer Science and Engineering, 15 credits.

Halmstad May/2023
Shahm Abdulrazzak, Tor Mattsson
Sammanfattning


Detta projekt har potential att förbättra rättningsprocesser i programmeringskurser genom att testa inlämningar på ett systematiskt och likvärdigt sätt och genom att vara ett mer effektivt sätt att bedöma studenternas inlämningar. Verktyget skulle kunna integreras med en Learning Management System för att även automatisera nerladdning av inlämningar och uppladdning av resultat.

Keywords:
Abstract

The process of grading code submissions in programming courses is time-consuming and error-prone. To address this issue, we propose a project that automates the testing and grading process for Java code submissions at Halmstad University. Our approach leverages property-based testing using the JUnit Quick-Check library to generate tests and test reports automatically.

The tool we demonstrate in this project is designed to provide students with immediate feedback on their code submissions and reduce the workload for instructors. By automatically generating tests and test reports, our approach ensures that code submissions meet the required specifications and are free from common errors. The com.pholser library is utilized to implement the property-based testing approach.

We believe that our project has the potential to improve the grading process for programming courses and provide a more efficient and effective way of assessing student code submissions. Our implementation can be easily extended to support other programming languages and can be integrated with existing learning management systems to provide a seamless experience for instructors and students alike.

Keywords:
Acknowledgments

We would like to express our sincere gratitude to the individuals who have made significant contributions to the completion of this project:

First and foremost, we extend our thanks to our supervisor, Veronica Gaspes, for her invaluable insight, guidance, and unwavering support throughout the entire project. Her expertise and continuous assistance have been instrumental in shaping the direction and quality of our work.

Additionally, we would like to acknowledge and express our appreciation to Professor Wojciech Mostowski for his exceptional teaching in object-oriented programming. The knowledge and skills imparted during his classes have proved to be immensely valuable, playing a pivotal role in the successful execution of this project.
Table of contents

1. Introduction .................................................................................................................. 7

2. Background .................................................................................................................... 9
   2.1 Property-based Testing Libraries .............................................................................. 9
   2.2 Similar Tools ............................................................................................................ 10
      2.2.1 CodeCheck ....................................................................................................... 10
      2.2.2 Nb-grader ......................................................................................................... 11

3. Methods ......................................................................................................................... 13
   3.1 Theory ........................................................................................................................ 13
      3.1.1 White- and black box testing ........................................................................... 13
      3.1.2 Property-based Testing .................................................................................... 13
      3.1.3 JUnit QuickCheck ............................................................................................ 14
   3.2 Methodology ............................................................................................................. 15
      3.2.1 Sub-goals .......................................................................................................... 15
      3.2.2 Project management tools ................................................................................ 15
      3.2.3 User-friendly environment using APIs .............................................................. 17
      3.2.4 Class Generator<T> ......................................................................................... 18
      3.2.5 Optimizing for Scalability and Performance ..................................................... 19
      3.2.6 Visualization of the test results ........................................................................ 19
      3.2.7 @Annotaions .................................................................................................... 20
      3.2.8 Support classes ............................................................................................... 21
      3.2.9 Object-oriented Programming ......................................................................... 22
      3.2.10 API and integration of LMS ............................................................................ 22

4. Result ............................................................................................................................. 23
   4.1 Breaking down the project ...................................................................................... 23
   4.2 Retrieving data from the LMS ............................................................................... 23
   4.3 Handling the zip files and compilation .................................................................. 23
   4.4 Running the PBT ..................................................................................................... 24
   4.5 Creating the text report ......................................................................................... 26
   4.6 Plagiarism ............................................................................................................... 27
   4.7 Final product summary ......................................................................................... 28
   4.8 Limitations .............................................................................................................. 29

5. Discussion ..................................................................................................................... 31
   5.1 Technical Achievements ......................................................................................... 31
      5.1.1 Implementation of PBT ................................................................................... 31
      5.1.2 Customization of the Testing Framework ......................................................... 31
5.1.3 Scalability of testing ......................................................... 31
5.1.4 Effective bug detection .................................................... 31
5.1.5 Support classes............................................................... 31
5.2 Results Related to the Research Questions .......................... 31
5.3 Societal Aspects ................................................................ 32
6. Conclusion ........................................................................... 35
References ................................................................................ I
1. Introduction

Code submissions are the most common way to test a student's knowledge in programming courses at universities. Hence, there are a lot of hand-ins to go through for the teachers. Teachers need to manually review or run scripts to test each student's code submission to ensure that it meets the assignment's requirements. This is a time-consuming task, especially when dealing with a large number of submissions.

Marking code submissions involves making sure that the code compiles, that the code calculates what is expected, and that the students do not share code. At Halmstad University code submissions are made through the University's learning platform (LMS) Blackboard. It takes a long time for teachers to download student submissions, compile them, execute them, and review the code for each student submission. The purpose of the project is to create a tool that will speed up the correction process of student submissions so the teachers can save time and focus on more important things. This project is relevant since an auto-correcting tool for submissions would facilitate the teacher's work.

The software produced in this project will be used to test the Java submissions of students at Halmstad University. The software will take the student's code and generate a set of test cases based on the properties specified by the instructor. The software will compile all students’ code, run the tests, possibly check for plagiarism among submissions and report any compilation errors, failed test cases, and possibly identified plagiarism to the instructor. This will allow the instructor to identify any bugs quickly and easily in the student's code and provide feedback to the student. The instructor will be able to download all submissions and the software will check all of them and generate reports for each of them.

By communicating with the customer, in this case, the teachers of the courses, the project can be created to satisfy their needs to accelerate the correction process of student submissions.

The problem statements to answer are how can property-based testing be integrated into an auto-grading system, and what are the benefits and limitations of this approach?
2. Background

When marking code submissions teachers must make sure that the code submitted by the students compiles, that the code calculates what is expected, and that the students do not share code. Automating parts of this process is what this project attempts. In particular, the project explores how to use Property-based testing to help automatize the process of checking that the code submitted by the students calculates what is expected. This section reviews the tools that can be used to leverage Property-based testing and some tools that have already been implemented to aid in the process of marking code.

2.1 Property-based Testing Libraries

JUnit-QuickCheck [1], QuickCheck [2, 3, 4], ScalaCheck [5], Hypothesis [6], and JQwik [7] are all libraries for property-based testing. They all generate random inputs for a test and check that the output of the test satisfies certain properties.

QuickCheck was originally implemented in Haskell. PBT has historically been “The thing that QuickCheck does”. It was created by Koen Claessen and John Hughes in 2000 and has since been used and spread to many languages such as Erlang, C#, and Scala. It introduced and provided automatic ways of shrinking test cases, making it easier to understand and debug the cause of the failure. Other similar libraries that are used for different languages are JUnit-QuickCheck, ScalaCheck, Hypothesis, and JQwik.

ScalaCheck is a library for Scala and junit-quickcheck integrates the quickcheck library with JUnit in Java. QuickCheck and ScalaCheck have been around for longer and have a more mature ecosystem, while JQwik and Hypothesis are relatively newer. QuickCheck and ScalaCheck are more similar, as they both use a similar approach to property-based testing and have similar syntax. The hypothesis has a different approach, and its syntax is more different from the first two. junit-quickcheck is an integration of quickcheck with JUnit and allows for quickcheck tests to be run within JUnit.

Hypothesis is a library from the programming language Python. It’s used for creating unit tests that are simpler to write compared to other PBT libraries. Even though it’s simpler, hypothesis is still powerful when run, finding edge cases in the code that one wouldn’t have thought to look for. Hypothesis is stable, powerful, and easy to add to any existing test suite. QuickCheck and ScalaCheck focus on generating random inputs, while Hypothesis focuses on generating inputs that are as small as possible while still revealing bugs.

JQwik is a Java library for property-based testing, while QuickCheck and Hypothesis are libraries for Haskell and Python, respectively. The main purpose of JQwik is to bring PBT to the Java virtual machine. The library’s focus is mainly on Java and Kotlin.

The use of JUnit-QuickCheck allows for seamless integration with JUnit, which is the most widely used Java testing framework. This means that
developers who are already familiar with JUnit can easily get started without learning a new framework. It allows for customizable generators, i.e. developers can create their custom generators for properties, which can be useful for testing specific types of data or edge cases. Furthermore, it's one of the most used within the area, hence has a strong contributing community. This means that plenty of resources and help are available if issues occur. For these reasons, it was decided to proceed to use the library JUnit-Quick-Check in the development of this project.

There are however some downsides and limitations to using JUnit-QuickCheck. Unlike ScalaCheck, it does not have as robust support for shrinking. This can make it more difficult to find the minimal input that causes the program to fail. Lastly, it’s less expressive compared to other property-based testing frameworks, which can make it more difficult to express the properties and supposedly behavior of a function. By the looks of it, this does not seem to limit us in this project.

2.2 Similar Tools
Grading programming assignments manually is a very time-demanding process, which is why auto-grading tools for programming submissions are being developed. Some of these auto-grading tools are CodeCheck [8, 9] and nb-grader [10].

2.2.1 CodeCheck
CodeCheck is an anonymous, author-friendly auto-grader. It is optimized for simple programming assignments that provide practice and build confidence. This software allows the teacher to specify a function and properties for the given function. It's an educational tool that easily makes it clear what part of the code for a given function should be visible to the students, and where they should write their code. This is done by some simple “//Commands”, i.e. //HIDE and //EDIT. The //HIDE command makes specific parts of the code invisible to the students, and the //EDIT command creates specific areas where they’re supposed to write their code. Moreover, it makes it clear where the teacher is supposed to write her code. This is a way of pointing out or highlighting parts of the relevant code to the students. It provides support for the existing PBT libraries and allows for user-friendly interfaces. It can be integrated into different learning management systems (LMS) such as CANVAS, Moodle, and Blackboard.

The teacher can then choose if they want to use unit testing, auto-generated tests, or other testing options and how the results are presented to the students. After the students submit their code they will receive instant feedback on the performance of their code, and how many of the criteria they fulfilled.

When a teacher uses CodeCheck to assess students' code assignments, they typically must perform the following tasks:

1. Create assignments: The teacher has to create assignments that contain instructions and requirements for the students' code assignments.
2. Release the assignments.
3. Collect the assignments.
4. Set up the code check tool: The teacher has to set up the code check tool with the requirements and tests specified in the assignment. This involves creating test cases that check whether the students’ code meets the requirements and produces the expected output.
5. Run the tests: The teacher can then run the tests using the code check tool to assess the students' code assignments. The tool will provide feedback on whether it meets the requirements and produces the expected output.
6. Provide feedback: Once the testing is complete, the teacher can provide feedback to the students by releasing the test results and any additional feedback.

The tool developed in our project will have some similarities with CodeCheck, mostly when it comes to the teacher creating test cases that check if the students’ programs pass the assignment.

2.2.2 Nb-grader
 Nb-grader is a command line tool and web application for creating and grading Jupyter Notebook assignments. It allows instructors to assign and grade notebooks, while also allowing the students to submit their work and receive feedback on their progress. It includes functionality for creating assignments, distributing them to students, collecting submissions, and providing feedback to students. It also provides support for automated testing of code and integration with other tools such as code coverage and style checkers.

The order nb-grader works in are as follows:
1. Write the instructor version (with answers)
2. Auto-generate student version (without answers)
3. Release The assignment to students.
4. Collect submissions from students.
5. Auto-grade submissions.
7. Generate feedback.

When a teacher is using nbgrader, they typically have to perform the following tasks:
1. Create assignments: The teacher has to create assignments in Jupyter notebooks that contain instructions and questions for the students.
2. Convert the notebooks: The teacher then has to convert the notebooks into a format that the nbgrader can grade. This has parts like removing the solutions to the questions and replacing them with code cells that grade the student's answers.
3. Release the assignments.
4. Collect the assignments.
5. Grade the assignments: The teacher can then use nbgrader to grade the notebooks submitted by the students. This involves running the grading code cells that check the students’ answers against the expected solutions.
6. Provide feedback: Once the grading is complete, the teacher can provide feedback to the students by releasing the graded notebooks with the solutions and feedback.
3. Methods

3.1 Theory

3.1.1 White- and black box testing
There are two major approaches to generating test cases for testing software: white-box testing and black-box testing.

White Box Testing involves examining the code and picking test cases that to some extent execute interesting parts of the code. Based on knowledge of the code structure, tests are identified so that when applied different code segments get tested. Some upsides are that: the method is useful for identifying internal flaws, such as coding errors and logical faults. It’s more comprehensive to write tests to cover the complete program since the tests are easier to apply to individual segments of the code. However, there are some downsides: it requires a deep and thorough understanding of the structure of the program, and trying to cover most of the code segments requires enormously many test cases [11].

Black Box Testing involves understanding the specification and the input and output domains to generate test cases. Without having any insight into how the code to be tested is built, tests are created from the outside. In other words, testers treat the software as a ‘black box’. Only focusing on the inputs and outputs of the program. Some upsides are that it is more suitable for testers who lack knowledge of how the software works. It tests the software system based on a user’s perspective. There are however some negative aspects of black box testing: testers might miss issues that could have been identified by looking at the code. It tests the program as a whole, which means it’s harder to isolate the cause of defects in the code. And finally, tests may be redundant if the conditions that are being tested are overlapping [11].

Overall, the differences between the two testing methodologies make them effective when applied to different projects depending on the purpose of the testing and the program itself.

3.1.2 Property-based Testing

A technique that is used in this project is Property-based testing, which will later be referred to as PBT [12]. PBT is a technique where test cases are generated automatically, based on properties that the code under test should satisfy. These properties can be seen as partial specifications of what the software is expected to do. PBT tries to combine the intuitiveness of tests with the effectiveness of randomized, generated test data. This is done by specifying properties that the code should satisfy, and then the testing library generates random inputs that should satisfy those properties. The advantage of this approach is that it can find bugs that would be difficult to find with traditional manual testing, as it can generate inputs that are unlikely to be considered by a human tester.
Property-based testing is a software testing technique where instead of testing a specific input-output combination, you test the behavior of the system or program under a set of properties or rules that it should follow [2].

In PBT, one can define a set of properties that the system should satisfy, and then generate random inputs that are used to test a piece of software against these properties. The test framework generates many test cases, often in the order of thousands, to test the properties and find potential edge cases and bugs that would be difficult to find using traditional unit testing techniques.

The approach is useful for testing complex systems or systems with many possible input combinations. By testing properties instead of specific input-output pairs, PBT can often uncover subtle issues or edge cases that might be missed with traditional testing methods.

PBT is commonly used in functional programming languages such as Haskell, but it can be used in other programming languages such as Java and Python as well. In this project, PBT in Java is of the most interest. There are several popular libraries and tools for property-based testing, including QuickCheck, Hypothesis, and ScalaCheck.

An example where PBT can be used is when one might want to test a function that sorts integers in ascending order. Instead of giving specific input integers to be sorted by the function and expecting a specific output, PBT generates random input integers to be sorted by the function, and the output is then tested to make sure that it’s in ascending order to make sure that the function does what it is intended to do.

3.1.3 JUnit QuickCheck

One of the important tools used in this project is JUnit-QuickCheck. JUnit-QuickCheck is a library for the JUnit testing framework that allows the user to generate and execute randomized test cases for the program to be tested. It is based on Property-based testing, which involves specifying the properties that a Java program should have and then generating test cases to check those properties.

With JUnit-QuickCheck, the tester can define a set of constraints and properties that the code must satisfy, and then let the library generate random input values to test those constraints and properties. In our project, we imagine the teacher producing the properties for the code that the students must write. This can help the teachers discover bugs and edge cases that they might not have thought of when manually writing test cases.

The library provides a set of annotations and utilities that allow the teachers to easily integrate randomized testing into their JUnit test suite. For example, they can annotate a test method with “@Property” to indicate that it is a Property-based test, and use the Gen class to generate random input values.
JUnit-QuickCheck is a useful tool for testing complex and difficult-to-reason-about code, as well as for finding edge cases and unexpected behavior in the code.

An example where JUnit QuickCheck is used in a tool is Apache Cassandra which is a distributed NoSQL database. The tool uses the PBT library JUnit QuickCheck to test its data access layer. The data access layer is responsible for writing data to the disk and reading it. The PBT library helps the developers at Cassandra ensure that their code can handle a wide range of input values, and edge cases and that it satisfies the properties defined for its data access layer.

3.2 Methodology

3.2.1 Sub-goals
The project was broken down into multiple sub-goals so that it’s easier to develop and test.

The sub-goals of what the tool should include are things such as being able to handle zip folders and unzipping them, iterating through a folder, detecting Java classes, compiling them, and running PBT tests on the Java classes that successfully compile. The last optional sub-goal that should be included in detecting plagiarism between students is to be able to know if students have copied each other’s code. Another important sub-goal that must be included in this tool is to create a Txt file that includes details of the results of how the correction process went.

Since assignments can be different, it is not meant to include written tests and data generators in the tool but to give the teachers the chance to easily write and implement their own tests and data generators for whatever assignments they’re expecting the students to submit. It should be easy for the teachers to implement those tests and generators to save as much time as possible.

The tool is developed to automate parts of the correction process of programming assignments which will also reduce the time spent on manually checking and correcting these assignments.

Another optional sub-goal is to integrate the tool with the university’s LMS (learning management system) so the teachers can automatically grade the submissions without needing to download them locally on their machines first.

3.2.2 Project management tools

To be able to facilitate the use of existing libraries and keep structure between source code and test code a management tool was needed. Maven is a build automation and project management tool [13]. It helps and provides ways to test, package, and deploy Java libraries as well as to manage the different dependencies. Dependencies in a Java project are external libraries that the project’s code depends on to run or compile. Maven manages the project
dependencies by downloading them from remote repositories and placing them on the classpath. To know which dependencies to use, examples were followed from JUnit Quick-Checks official page together with other JUnit Quick-Check examples. The “com.pholser” library was most widely used in this project [14].

Different packages were created to maintain a structure between the test code and the main code. Within these folders, subdirectories for different types of code and resources were used.

Overall, for someone with no previous experience with Maven or PBT, the setup for everything to work might not be effortless. To facilitate the threshold of creating these PBT, a ‘ContentCreator’-class was created. Its’ purpose is to serve as an adaptable template. When instantiating the class object, it takes as in arguments: generatorType, packageName, className, GeneratorClass, and Trials. By creating an object of the class, the user can seamlessly create a test class, create a generator or set up the pom file by using these methods: ‘.createTestClass’, ‘.createGeneratorClass’ ‘.createAPomFile’.

As a user of the auto grader, what is left to do? After the “contentCreator” class creates the .java-files: testClass and generatorClass, all that’s left is to specify the tests themselves inside the testClass and to specify the behavior of the tested class in the generatorClass. By passing ‘generatorType’, ‘packageName’, ‘ClassName’, ‘generatorClass’, and ‘trials’ as arguments to the ‘ContentCreator’- object, it will generate the test files and Java code necessary. The areas left to change are pointed out and, notable is also that all necessary libraries are dynamically imported. See code example 1,2,3.

```java
public static void main(String[] args) throws IOException {
    ContentCreator example = new ContentCreator(...);
    example.createGeneratorClass();
    Example.createTestClass(...)
}
```

*Code example 1: How to set up all tests using content creator.*
public class ClassName extends Generator<generatorClass> {
...

    @Override
    public ClassName generate(SourceOfRandomness random, GenerationStatus status) {
        //Customize the behavior of your Generator
        return new ClassName();
    }
}

Code example 2: Visualization of how the created generator file looks like.

@Property(trials = 100)
public void test3hugo(@From(ClassName.class) ClassName obj) {
    //Customize the behavior of your test
    Boolean test = false /* Replace false with your test here*/;
    ...
}

Code example 3: Visualization of how the created Test file looks like.

3.2.3 User-friendly environment using APIs

In terms of a user interface for JUnit Quick-Check, there is not a graphical user interface (GUI) per se, but rather a set of APIs that provides a programmatic interface for defining properties and generating test data.

To define a property in JUnit QuickCheck, you simply annotate a method using the `@Property` annotation, followed by the test logic in the body of the method. Junit QuickCheck provides a set of built-in generators for generating test data, and you can also define custom generators if needed. See 3.1.4.

To generate and run property-based tests, you can simply run the test class containing the annotated properties as you would with any other Junit test class. By passing the `@From` annotation followed by the GeneratorClass.class and then the Class c as arguments of the testFunction, Junit QuickCheck will use the given generator and automatically generate random test data and execute the property-based tests. See code example 3:

Using the `@SuiteClasses` annotation simplifies things by allowing multiple tests to be run at once by a single command. Rather than having to run each test class separately, they can be grouped and executed all at once. This saves time and makes it easier to manage your test, particularly when the amount of tests increases. Additionally, using a test suite allows for structural support. The tests can be organized into local groups, such as by functionality or
module, making it easier to understand and maintain structure within the project.

When using the ‘com.pholser’ library for PBT, you can create a suite of tests by defining multiple test classes and annotating them with ‘@RunWith(PropertyTestRunner.class)’ and ‘@TestProperties’ annotation. Once you have defined your test classes you can use the ‘@SuiteClasses’ annotation to group them and run them as a suite. In this example, the ‘MyPropertyTestSuite’ class serves as a container for the three test classes which are specified in the ‘@SuiteClasses’ annotation. When you run this test suite, JUnit will execute all the tests in the three classes. See code example

```java
@RunWith(Suite.Class)
@SuiteClasses(
    PropertyTest1.class
    ...
    PropertyTestN.class
)
...
```

*Code example 4, How Property tests are integrated into one SuiteClass.*

So, while JUnit QuickChest does not provide a GUI per se, it does provide a convenient and intuitive programmatic interface for defining properties and generating test data which can be considered a form interface for the framework [16].

### 3.2.4 Class Generator<T>

Generators are a crucial component of PBT because they enable the automated generation of inputs that conform to the properties being tested. Generators can make testing more reusable and optimized for scalability in several ways [17].

Firstly generators allow for a wider range of input values, compared to when creating objects manually. This is particularly important when testing complex programs or testing edge cases that may be difficult to identify manually. By restricting and defining the generator, it generates a diverse set of inputs and ensures that the program behaves correctly across a wide range of scenarios.

Secondly, by defining a generator, it can improve the scalability of testing by automating the process of generating inputs. This is particularly useful when testing a large or complex program, where manually generating inputs may be impractical and tedious. By automating the input generation process, generators can significantly reduce the time and effort required to test a program.

Finally, generators make the testing more reusable by allowing the reuse of the same test inputs across multiple tests. This is particularly useful when
testing multiple properties of a program, as it allows testing of each property with the same set of inputs, thereby reducing the risk of introducing errors or inconsistencies between tests.

An example of where the class generator can be used is for instance when testing a Java class that represents a complex number. In that class, the students could submit different functions such as functions that add, multiply or subtract two complex numbers. To test these functions, data that represents complex numbers must first be created, and by using the generators class, these complex numbers can be created automatically and then the functions can then be tested with PBT.

In summary, understanding how generators work and function is crucial to make the PBT more reusable and optimizing it for scalability.

3.2.5 Optimizing for Scalability and Performance

To optimize the performance, one of the key factors is to have effective generators [3]. Once the property test begins running, each iteration produces a new class object via the generator. Hence, the code complexity has a big compact over the overall runtime. The property tests of course also have an impact on the performance, however, the functionality is of greater interest. In John Hugestalk’s "How to Specify It" [1], John Hughes emphasizes the importance of focusing on the functionality of the software being developed, rather than on the complexity of the tests that verify the correctness of that software. He argues that if the tests themselves become too complex, they can become a maintenance burden and may even introduce bugs, defeating their purpose. Instead, he suggests that tests should be kept as simple as possible while still providing good coverage of the desired functionality.

3.2.6 Visualization of the test results

The com.pholser library doesn’t have support for any tool to present data or results after the execution of the test classes. However, this meant a custom-made TestReporter class had to be created instead. A combination of TestReport-, TextFileHandler- and WriteToFile- classes was used to create and write a TestReport.txt file.

To find potential errors, expressions such as ‘assertTrue’ and ‘assumeThat’ were imported ‘import static org.junit.Assert…’.

The ‘assumeThat’ method is used in PBT to conditionally include or exclude test data based on a given predicate. With the generators, test data is generated automatically based on the properties of the generator. However, not all generated test data may be valid or relevant for the particular test case being executed. The ‘assumeThat’ method provides a way to filter out unwanted test data by specifying a predicate that must be satisfied for the test to continue executing with the data.
The ‘assertTrue’ method in JUnit is used to assert that a given condition is true. When used in PBT, ‘assertTrue’ can be used to validate generated test data by checking if certain properties hold. It can be used to validate various properties of the generated test data, such as range, inequalities, or specific values. By asserting that these properties hold for the generated data, the confidence increases the correctness of the tested code.

Once all trials have been executed, the test report is generated in a text file. Due to the number of trials, the test reporter will generate results in the text file equal to the number of trials. However, this is not a suitable or visually appealing approach to a result report. To fix this, a textFileHandler-Class was implemented, to scan through the first text-file, and interpret the outcome of the tests. The results are then presented as the number of completed tests out of the total amount of tests, and if any of them fail, the argument with the failed values is also presented in the visual report.

3.2.7 @Annotations
To find potential errors, expressions such as ‘assertTrue’ and ‘assumeThat’ were imported ‘import static org.junit.Assert…’.

The ‘assumeThat’ method is used in PBT to conditionally include or exclude test data based on a given predicate. With the generators, test data is generated automatically based on the properties of the generator. However, not all generated test data may be valid or relevant for the particular test case being executed. The ‘assumeThat’ method provides a way to filter out unwanted test data by specifying a predicate that must be satisfied for the test to continue executing with the data.

The ‘assertTrue’ method in JUnit is used to assert that a given condition is true. When used in PBT, ‘assertTrue’ can be used to validate generated test data by checking if certain properties hold. It can be used to validate various properties of the generated test data, such as range, inequalities, or specific values. By asserting that these properties hold for the generated data, the confidence increases the correctness of the tested code.

There are however a lot more ways of restricting and managing the test data to behave in certain ways. Some examples are: ‘@Assume’, ‘@Seed’, ‘@With’, ‘@NotEmpty’, ‘@NotNull’, and ‘@InRange’. [13, 18, 19]

After the ‘@Property’ annotation in a property-based test, the “(trial = x)” statement indicates the number of generations from the given generator class by changing the value of x. This means that the number of times the test will be run also equals x. To avoid x- iterations of reports in the final visual representation, a scrap file is used. The scrap file contains all information about each iteration and the test results. A scanner then scans through the scrap file analyzing and finalizing a visual representation of the test output. See Figure 1.
Figure 1: Final Txt file containing the result of the operations.

3.2.8 Support classes

When teachers download students’ submissions from the LMS, they are all downloaded as zip folders. For that reason, the tool must be able to unzip folders before anything else. To do that, a ZipInputStream is used. ZipInputStream is a Java class that implements an input stream filter for reading files in the ZIP format. To extract files correctly from a ZIP folder, a ZipEntry is used. ZipEntry is a class that is used to represent a ZIP file entry. After unzipping all submissions, the tool has to compile all programs. To compile Java programs, a ProcessBuilder is used. ProcessBuilder is a class that is used to create operating system processes. When a program has a compilation error, that error is caught with a BufferedReader. BufferedReader is a Java class that simplifies reading text from a character input stream. BufferedReader buffers characters for efficient reading of characters, arrays, and lines.

After the program is done compiling all the Java files, they are ready for the PBT. However, to run the correct PBT on the correct submission class everything needs to be in the right place. The tests are based on generating multiple class objects, which is based on the class generator, which itself is based on the class submission. For the generator class to find the correct submission class, a placeholder class has been created. The submission classes will then be copied over to the placeholder and the generator will
attempt to create objects of the given class. It’s those class objects which the tests will be invoked upon.

Those tests will generate a text file containing information on how the whole process went. The text file will contain the student’s name, information on how the compilation process went, and if the students’ programs passed or failed the tests and which tests were invoked. If a failure occurs, it’s shown with which arguments it failed.

To maximize the efficiency and let the PBT serve its’ purpose, a lot of trials have to be done. This means that the text file previously mentioned will contain information about each single trial. This means that the text file will work as a scrap file.

To create and edit the text file, a FileWriter is used. FileWriter is a Java convenience class for writing character files. FileWriter is meant for writing streams of characters. This combined with a TextFileHandler which scans the scrap file and analyzes its content works as a complete functional text file reporter.

3.2.9 Object-oriented Programming
It's important to have some knowledge about object-oriented programming (OOP) when developing the tool in this project. OOP offers multiple benefits such as modularity, which means that OOP helps to divide a large program into smaller, more manageable pieces (objects) that can be developed and tested separately. This was heavily useful in the development so far. Examples of objects that have been created so far are a folder unzipper and a compiler.

3.2.10 API and integration of LMS
By directly retrieving the data from the LMS, the work from the teachers' side would become even easier. Retrieving data from the LMS would only require GET functions for the zip files from the eventual Rest API. The LMS used by Halmstad University is blackboard. Furthermore, they provide documentation to their Rest API [20]. However, to access the specific zip file you either need authentication or an admin login. This could technically still be done, by using a test environment. This would nevertheless not objectively help the work for the users.
4. Result

4.1 Breaking down the project

The project can be broken into different segments, and by looking at them one by one, it is easier to visualize the results of the project. Breaking down the project also helped with developing the project and easily finding errors and bugs in the code. The project was developed with the object-oriented programming aspect in mind. The tool was broken down into many different classes that have different functions. These classes include a class that compiles Java classes, another class that unzips a folder, a third class that iterates through the contents of a folder, and a fourth class that creates a `Txt` file.

Another way that this project was broken down was into different packages that contained different kinds of classes. These packages include a package that has all the exercises that the teacher is expecting to be submitted by the students, another package that has all the different data generators, and a third package that includes all the test classes that will be edited by the teacher to test students’ submissions.

4.2 Retrieving data from the LMS

As mentioned earlier in 3.2.10, there are no APIs included in this project. In other words, the data must manually be downloaded from the given LMS. There was work done towards creating a Rest API but was restricted by the LMSs (Learning Management Systems) authentication.

4.3 Handling the zip files and compilation

After the data is retrieved, the zip files must be handled. The program takes two string inputs, one for the path of the zip folder containing all students’ submissions, and the second one for the direction of the unzipped folder. After the program receives the inputs, it unzips the first folder containing the submissions, and then iterates through the unzipped folder and unzips all students’ folders (one for each student). The tool then iterates through all student submissions’ folders and compiles all Java programs. At the end of all this, the tool generates one text file that contains information about how this entire process went. The text file includes all students’ names and states how the compilation went for every Java program that is submitted.

By looking at Figure 2 one could see what the text file would look like. Figure 2 shows an example of the text file when the students’ programs compile successfully. In this example, each student has 3 different programs, E1.java, E2.java, and E3.java. The text file shows the student’s name (that’s represented by the first 3 characters of the student’s name plus the first 3 characters of the student’s last name and the year the student started studying at the university) and all submitted programs by the student compiled successfully and that is presented with the message “Compilation successful for file: “ followed by the file name and path.
By looking at Figure 3, one could see how the text file would look when one of the submitted programs has a compilation error. The text file still shows the name of the student and shows the programs that compile successfully. When one (or more) program doesn’t compile, it shows where the program failed to compile and what the syntax error is. In the example in Figure 3, the students wrote “asdfsdf” in their program and the compiler returned that this is not a statement and that it expected a semicolon at the end of the statement.

4.4 Running the PBT
After a Java file is successfully compiled, it’s copied to the project and tested with the PBT test for that specific Java class.
To test the test environment itself, an example class with properties had to be created, in this case: Complex Number. The class is a class representation of a complex number that imitates a student submission. It is presented as a real part number and an imaginary part number and has some methods or operations which apply to the Complex Number.

When testing properties for the complex number, the absolute value of vectors presented on the complex table was tested. One property in particular, for any combination of values of the real and imaginary part, all vectors must have a length that is greater than one. Moreover, all vectors multiplied by the complex number $I$ are supposed to rotate counterclockwise by 90 degrees. Hence, the length or absolute value of the vector should remain the same. To ensure the performance of our complex number they were put up for the following tests. The latter test behaves incorrectly. See code example 5, figures 4, 5. It becomes clear which test failed and which passed, and with what arguments it fails. A way to develop this further could be bypassing the shrunken version of the failed test case. See the source for more information about shrinking [16].

```java
@Property(trials = 10)
public void testComplexNumberNorm(@From(ComplexNumberGenerator.class) ComplexNumber c) {
    ComplexNumber I = ComplexNumber.I;

    //Correct way of testing norm
    Boolean test = (c.norm() == c.times(I).norm());

    //Incorrect way of testing norm
    Boolean test = (c.norm() == c.times(I).norm()+1);

    ...
}
```

*Code Example 5: Ways of testing.*

*Figure 4: Report generated from the first test in Code Example 5 ‘correct way of testing norm’, test passed.*
4.5 Creating the text report.

Two text files are created during the PBT. One temporary text file, see Figure 6, and the other containing the summary of the temporary file, see Figure 7. When appending the string from the summary text file combined with compilation messages you get the complete report for all submissions, see Figure 8.

Figure 5: Report generated from the second test in Code Example 5.

Figure 6: the temporary file containing all information about each trial.

Figure 7: the final visual representation of all trials on one test after scanning the temporary file.
4.6 Plagiarism

Due to a lack of time, plagiarism detection is not explicitly implemented in this project. However, a K-gram plagiarism detection tool [21] is available and can be used manually.

The k-gram plagiarism detection algorithm is a text similarity detection method that works by comparing k-length sequences, or k-grams, of two texts. It is often used to detect cases of plagiarism in documents, where one author may have copied part of another author's work without proper citation or attribution. However, the tool can still be used to compare Java classes.

The algorithm works by breaking each document into smaller sequences of characters of length k that often overlap. These sequences are then compared between two documents and the number of matching sequences is counted. The more matching sequences, the higher the probability of plagiarism.

Since there wasn’t enough time to develop the plagiarism detection tool for this project, a finished implementation was downloaded and used [21]. To optimally use this tool, the user has to use the best numbers for “k” and “d”. To find the best suitable numbers for “k” and “d” for this project, a small experiment was done. First, two identical documents were compared with different k- and d-values, all different values gave the same answer of 100% match between the two documents. When copying code, students often change the order in which functions are written. For that reason, the second part of this experiment was to compare two identical documents but move the functions around in one of the two documents and test it with different k- and d-values. The result was still the same 100% match with different values. The third test that was made was to delete some of the functions in the first document and compare it to the second document. In this test, it was obvious that the result was more accurate when the k-value was set to 3 and the d-value was set to 1000. The test result was reasonable since these documents that are being compared are rather short and a smaller k-value may be more effective for shorter documents while a larger k-value may be more appropriate for longer documents.
The value of k can vary depending on the application and the length of the compared text. Larger values of k may be more suitable for longer documents, while smaller values of k may be more effective for shorter documents. That’s why, to detect plagiarism between Java classes, it’s advised to set the k value to 3.

In this context, “d” stands for “document distance” or “edit distance” between two documents.

The document distance is often used as a metric to compare the similarity of two documents that have been represented as vectors of k-gram frequencies. By calculating the document distance between two documents, we can determine how closely related they are in terms of their content and structure. A lower document distance indicates greater similarity between the two documents, which is why it’s suitable for this project to use the d-value of one thousand.

The tool developed in this project will sort all student submissions into folders, so it is easier to test them for plagiarism. For example, if each student submits “E1.java”, “E2.java” and “E3.java”, the tool will create a folder named “E1” that contains all “E1.java” classes submitted by the students, another folder named “E2” that contains all “E2.java” classes, and a last folder named “E3” containing all “E3.java” classes. This makes it easier for the teacher to manually compare these documents with each other using the K-gram plagiarism detection algorithm to flag the student that has possibly copied code from each other.

By looking at Figure 9, one could see an example of 10 students’ assignments being compared to each other. In this specific example, 9 of the 10 students have the same code. The first student had an 81% match with the other students while students 2-10 had a 100% match with each other. This will help the teacher to easily spot the students that have possibly copied each other’s code.

By looking at Figure 9, one could see an example of 10 students’ assignments being compared to each other. In this specific example, 9 of the 10 students have the same code. The first student had an 81% match with the other students while students 2-10 had a 100% match with each other. This will help the teacher to easily spot the students that have possibly copied each other’s code.

![Figure 9: Result matrix when running plagiarism detection tool.](image)

4.7 Final product summary

Due to security authentication reasons, the tool can’t retrieve students’ submissions directly from the school’s LMS. After manually downloading the students’ submissions from the school’s LMS, teachers will have to write their own tests and data generators for their tests. This is implemented so that the teachers have to spend as little time as possible writing these tests and generators. In the case of writing tests, the teachers must edit one line of code to test a specific function.
In the case of data generators, teachers might have to write a couple of lines of code depending on what they want to generate. How many lines of code the teachers have to write depends on how much data they want to generate.

After downloading the data, and writing the tests and generators, the tool is ready to be used. After all of that is done, the teacher only needs to give the path of the zip folder containing the submissions, and the tool will unzip all the folders and iterate through each student’s submission. When the iterator finds a Java file, it compiles it and if the operation succeeds, the tool runs the necessary test on the Java class. The result of the compilation process and test process is printed in the final text file with the correct student name who submitted these classes.

In Figure 1, in 3.2.7 one can see what the final text file might look like. In this example, the student has submitted 5 classes where 2 of them didn’t compile. When the compilation process is not successful, the tests are not run. However, when the compilation process is successful, the tests are run and a small report of how the test went is shown.

4.8 Limitations
The auto grader has some limitations when using it, this includes that it can’t test extremely simple programs such as programs written in the main function in Java. This is because these simple programs don’t require any kind of auto-generated data to correct them. These programs are primarily about learning how to use simple functions such as printing information to the console or understanding the differences between “int”, “string” and “double” for instance. While the auto grader is great for testing submitted programs such as “complex numbers”, “doubly linked list”, “node”, “stack” and so on, it is unable to correct more complex programs such as programs with a graphical user interface (GUI) or programs that use threads such as setting up a chat server with clients that can connect to the server and chat with other users. These classes don’t require data sets to be generated and tested and can’t be corrected using this auto-grading tool. Another limitation is when it comes to some students who name their functions something different than what the teacher specified, this could lead to the function not getting tested or corrected by the auto-grader. Another example where a function might be excluded from grading is if some students decide to change the type of data that the function returns (different than what was specified in the instructions).
5. Discussion

5.1 Technical Achievements

5.1.1 Implementation of PBT
Our project successfully implemented PBT using the .com pholser library in Java. This allowed us to automatically generate a large number of test cases based on a set of properties or specifications, rather than manually writing test cases and expecting a single output for a given input.

5.1.2 Customization of the Testing Framework
We were able to customize the .com pholser library to meet our specific testing requirements. For example, we developed custom generators. Customizable generators can be used for specific types of input.

5.1.3 Scalability of testing
Our project demonstrated the scalability of PBT, as we were able to generate and execute thousands of test cases in a relatively short period. This allowed us to test edge cases and find previously unknown bugs.

5.1.4 Effective bug detection
Demonstrating the feasibility and effectiveness of a new testing tool is challenging. One reason for this is that it can be difficult to compare the results obtained using different testing methods, as different methods have different strengths and weaknesses. The project was developed with consideration of object-oriented programming and that helps us and future users and developers to easily detect bugs in the code.

5.1.5 Support classes
The support classes that are needed for this project (described in sub-goals 3.2.1) such as unzipping folders, compiling classes, and creating a Txt file that contains the result of the operation are successfully implemented and are a crucial part of this tool. One thing that could have been done better is the way the students’ submissions are retrieved. Instead of having to manually retrieve the data, it could have been smoother to let the tool automatically retrieve the data from the school’s LMS. However, due to limitations when it comes to access and authentication, this was not possible to implement.

5.2 Results Related to the Research Questions

The problem statement lay: “How can property-based testing be integrated into an auto-grading system, and what are the benefits and limitations of this approach?”.

One thing to acquire and strive for is effectiveness and efficiency. It is hard to generalize the results obtained from the testing of an auto grader with PBT. The effectiveness of the approach greatly depends on factors such as the
complexity of the software being tested, the quality of the property specification, and the generators.

As previously stated, specifying accurate and relevant properties for a given program can be challenging, and incorrect or incomplete specifications can lead to inadequate testing. In other words, the limitation lies in the user's capability to specify the tests and generators. Moreover, since the testing is entirely automated, it may lack the ability to capture certain edge cases that a human tester might be able to identify quite easily.

Although it’s hard to measure ‘the success’ of a testing tool, by generating thousands of test cases you greatly increase the odds of finding potential bugs and detecting errors. With a high enough number of trials, automated testing surpasses the human capability of identifying errors.

When it comes to the results, one could argue that the number of tests is insufficient to ensure a fully functional Auto grader. However, a wider range of tests would indeed further test the class which is of testing. But, it would not test the auto grader itself. The tests are there to demonstrate that testing the properties of a class is possible. This means that the limitation lies in the boolean test, which tests the properties, and not the Auto grader itself. Testing one class in this case is enough to ensure that the auto grader works as intended. It's also enough to know what is needed to be provided by the teacher to correctly use the tool.

Overall, these technical achievements demonstrate the effectiveness and value of property-based testing in software development. They also highlight the key contributions of our project to the field of software engineering, including the development of a scalable and customizable framework for property-based testing in Java.

5.3 Societal Aspects

When correcting submissions, fairness is of the highest importance. Auto grading would help reduce the risk of human bias in the grading process. Since the tests are done by pre-determined criteria, it can ensure that all students are evaluated equally.

The grading should not be determined by who you are, or which teacher is correcting. The grading must only be based on the code submitted by the student. Furthermore, when the grading process becomes standardized, this would help to improve the consistency of the grading process, which means that it doesn’t matter which teacher grades which assignment, and all students will have to meet the same criteria that are set.

Standardization could help ensure consistency and fairness in the grading. This will prove to be even more important in high-stakes contexts, i.e. when the students’ future opportunities depend on the grading.
To be able to utilize the auto grader, the data of the submissions need to be collected. Whenever data is collected, there is potential for privacy concerns. E.g if the auto-grader collects data on the student’s performance, there may be concerns about how this data is used. Even more, who has access to it, and for what purposes it’s being used? It is important to address these issues, to protect the students’ privacy. This tool is developed for teachers and when they use it and get results back, they’re asked to not share the results with anyone who doesn’t have authority.

From an environmental aspect, the auto grader might be energy efficient. When grading manually, that may require a significant amount of computer power or other devices used by the grader. Reducing the time it takes for the grading process, most likely reduces the overall energy usage associated with grading.

Similarly, automating the grading process and digitally providing feedback might reduce the overall amount of papers being used. This could have a positive effect on the environment by reducing the number of trees that need to be cut down.

The same thing goes for the transportation aspect. By being able to work remotely, one could potentially reduce the amount of transportation required to physically transport assignments between students and graders.
6. Conclusion

The question remains: Is this a successful project?

Strengths:

- The auto grader uses property-based testing and contributes to the overall goal of improving software quality by enabling automated testing of software properties.
- The auto grader uses built-in methods and functions to provide an easy-to-use interface for the user.
- The auto grader using property-based testing contributes to the goal of reducing software development time by automating the testing process.
- The tool can handle zip folders and unzip them.
- The tool can also iterate through folders and compile Java classes that it detects.
- The tool will help teachers save time by automatically compiling and testing the students’ programs.
- The tool also brings a new fairness aspect to correcting students’ submissions since it will always test the same properties in the different students’ programs.

Demonstrating the feasibility and effectiveness of a new testing tool is challenging. One reason for this is that it can be difficult to compare the results obtained using different testing methods, as different methods have different strengths and weaknesses.

Furthermore, it is hard to generalize the results obtained from the testing of an auto grader with PBT. The effectiveness of the approach greatly depends on factors such as the complexity of the software being tested, the quality of the property specification, and the generators.

These are some of the weaknesses we were not able to prove but believe we managed to achieve.

Weaknesses:

- We were unable to prove that the auto-grader using property-based testing contributes to the goal of increasing software reliability by providing a more thorough and automated testing process.
- We were unable to prove that the auto-grader could successfully identify previously unknown software bugs, demonstrating the feasibility of the approach.
- We were unable to prove that the auto-grader testing increased the overall software reliability by catching more software defects, demonstrating its contribution to the project’s goal of improving software quality.
• To detect plagiarism, the algorithm must be run manually. This part could have been automated if we had more time.

In conclusion, we would argue that the project fulfilled most of the sub-goals. Moreover, we were able to point out the weaknesses where future projects may continue the development of a property-based testing auto-grader. This is why we consider this project a success!
References


