Introduction to testing

Learning Objectives

- Can tell about some issues caused by software bugs.
- You know what a stack trace is, the steps taken in troubleshooting, and can give textual test inputs to a Scanner.
- You know what unit testing is all about and you can write unit tests.
- You know about test-driven software development.

Let's take our first steps in the world of program testing.

Error Situations and Step-By-Step Problem Resolving

Errors end up in the programs that we write. Sometimes the errors are not serious and cause headache mostly to users of the program. Occasionally, however, mistakes can lead to very serious consequences. In any case, it's certain that a person learning to program makes many mistakes.

You should never be afraid of or avoid making mistakes since that is the best way to learn. For this reason, try to break the program that you're working on from time to time to investigate error messages, and to see if those messages tell you something about the error(s) you've made.

Software Error

The report in the address http://sunnyday.mit.edu/accidents/MCO_report.pdf describes an incident resulting from a more serious software error and also the error itself.

The bug in the software was caused by the fact that the program in question expected the programmer to use the International System of Units (meters, kilograms, ...) in the calculations. However, the programmer had used the American Measurement System for some of the system's calculations, which prevented the satellite navigation auto-correction system from working as inteded.

The satellite was destroyed.

As programs grow in their complexity, finding errors becomes even more challenging. The debugger integrated into NetBeans can help you find errors. The use of the debugger is introduced with videos embedded in the course material; going over them is always an option.

Stack Trace

When an error occurs in a program, the program typically prints something called a stack trace, i.e., the list of method calls that resulted in the error. For example, a stack trace might look like this:

Sample output

```
Exception in thread "main" ...
    at Program.main(Program.java:15)
```

The type of error is stated at the beginning of the list, and the following line tells us where the error occurred. The line "at Program.main(Program.java:15)" says that the error occurred at line number 15 in the Program.java file.

Sample output

```
at Program.main(Program.java:15)
```

Checklist for Troubleshooting

If your code doesn't work and you don't know where the error is, these steps will help you get started.

1. Indent your code properly and find out if there are any missing parentheses.
2. Verify that the variables used are correctly named.
3. Test the program flow with different inputs and find out the sort of input that causes the program to not work as desired. If you received an error in the tests, the tests may also indicate the input used.
4. Add print commands to the program in which you print out the values of the variables used at various stages of the program's execution.
5. Verify that all variables you are using are initialized. If they aren't, a NullPointerException error will occur.
6. If your program causes an exception, you should definitely pay attention to the *stack trace* associated with the exception, which is the list of method calls that resulted in the situation that caused the exception.
7. Learn how to use the debugger. The earlier video will get you started.

Passing Test Input to Scanner

Manually testing the program is often laborious. It's possible to automate the passing of input by, for example, passing the string to be read into a Scanner object. You'll find an example below of how to test a program automatically. The program first enters five strings, followed by the previously seen string. After that, we try to enter a new string. The string "six" should not appear in the word set.

The test input can be given as a string to the Scanner object in the constructor. Each line break in the test feed is marked on the string with a combination of a backslash

and an n character "\n".

```
String input = "one\n" + "two\n"  +
               "three\n" + "four\n" +
               "five\n" + "one\n"  +
               "six\n";

Scanner reader = new Scanner(input);

ArrayList<String> read = new ArrayList<>();

while (true) {
    System.out.println("Enter an input: ");
    String line = reader.nextLine();
    if (read.contains(line)) {
        break;
    }

    read.add(line);
}

System.out.println("Thank you!");

if (read.contains("six")) {
    System.out.println("A value that should not have been added to the group was added to it.
}
```

The program's output only shows the one provided by the program, and no user commands.

Sample output

```
Enter an input:
Enter an input:
Enter an input:
Enter an input:
Enter an input:
Enter an input:
Thank you!
```

Passing a string to the constructor of the Scanner class replaces input read from the keyboard. As such, the content of the string variable `input` 'simulates' user input. A line break in the input is marked with `\n`. Therefore, each part ending in an newline character in a given string input corresponds to one input given to the `nextLine()` command.

When testing your program again manually, change the parameter Scanner object constructor to `System.in`, i.e., to the system's input stream. Alternatively, you can also change the test input, since we're dealing with a string.

The working of the program should continue to be checked on-screen. The print output can be a little confusing at first, as the automated input is not visible on the screen at all. The ultimate aim is to also automate the checking of the correctness of the output so that the program can be tested and the test result analyzed with the "push of a button". We shall return to this in later sections.

Unit Testing

The automated testing method laid out above where the input to a program is modified is quite convenient, but limited nonetheless. Testing larger programs in this way is challenging. One solution to this is unit testing, where small parts of the program are tested in isolation.

Unit testing refers to the testing of individual components in the source code, such as classes and their provided methods. The writing of tests reveals whether each class and method observes or deviates from the guideline of each method and class having a single, clear responsibility. The more responsibility the method has, the more complex the test. If a large application is written in a single method, writing tests for it becomes very challenging, if not impossible. Similarly, if the application is broken into clear classes and methods, then writing tests is straightforward.

Ready-made unit test libraries are commonly used in writing tests, which provide methods and help classes for writing tests. The most common unit testing library in Java is JUnit , which is also supported by almost all programming environments. For example, NetBeans can automatically search for JUnit tests in a project — if any are found, they will be displayed under the project in the Test Packages folder.

Let's take a look at writing unit tests with the help of an example. Let's assume that we have the following Calculator class and want to write automated tests for it.

```
public class Calculator {

    private int value;

    public Calculator() {
        this.value = 0;
    }

    public void add(int number) {
        this.value = this.value + number;
    }

    public void subtract(int number) {
        this.value = this.value + number;
    }

    public int getValue() {
        return this.value;
    }
}
```

The calculator works by always remembering the result produced by the preceding calculation. All subsequent calculations are always added to the previous result. A minor error resulting from copying and pasting has been left in the calculator above. The method subtract should deduct from the value, but it currently adds to it.

Unit test writing begins by creating a test class, which is created under the Test-Packages folder. When testing the `Calculator` class, the test class is to be called `CalculatorTest`. The string `Test` at the end of the name tells the programming environment that this is a test class. Without the string Test, the tests in the class will not be executed. (Note: Tests are created in NetBeans under the Test Packages folder.)

The test class `CalculatorTest` is initially empty.

```
public class CalculatorTest {

}
```

Tests are methods of the test class where each test tests an individual unit. Let's begin testing the class — we start off by creating a test method that confirms that the newly created calculator's value is intially 0.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void calculatorInitialValueZero() {
        Calculator calculator = new Calculator();
        assertEquals(0, calculator.getValue());
    }
}
```

In the calculatorInitialValueZero method a calculator object is first created. The assertEquals method provided by the JUnit test framework is then used to check the value. The method is imported from the Assert class with the import Static command, and it's given the expected value as a parameter - 0 in this instance - and the value returned by the calculator. If the values of the assertEquals method values differ, the test will not pass. Each test method should have an "annotation" `@ Test`. This tells the JUnit test framework that this is an executable test method.

To run the tests, select the project with the right-mouse button and click Test.

Running the tests prints to the output tab (typically at the bottom of NetBeans) that contains some information specific to each test class. In the example below, tests of the CalculatorTest class from the package are executed. The number of tests executed were 1, none of which failed — failure in this context means that the functionality tested by the test did not work as expected. ->

Sample output

```
Testsuite: CalculatorTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.054 sec

test-report:
test:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Let's add functionality for adding and subtracting to the test class.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void calculatorInitialValueZero() {
        Calculator calculator = new Calculator();
        assertEquals(0, calculator.getValue());
    }

    @Test
    public void valueFiveWhenFiveAdded() {
        Calculator calculator = new Calculator();
        calculator.add(5);
        assertEquals(5, calculator.getValue());
    }

    @Test
    public void valueMinusTwoWhenTwoSubstracted() {
        Calculator calculator = new Calculator();
        calculator.subtract(2);
        assertEquals(-2, calculator.getValue());
    }
}
```

Executing the tests produces the following output.

Sample output

```
Testsuite: CalculatorTest
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.059 sec

Testcase: valueMinusTwoWhenTwoSubstracted(CalculatorTest): FAILED
expected:<-2> but was:<2>
junit.framework.AssertionFailedError: expected:<-2> but was:<2>
at CalculatorTest.valueMinusTwoWhenTwoSubstracted(CalculatorTest.java:25)

Test CalculatorTest FAILED
test-report:
```

```
test:
BUILD SUCCESSFUL (total time: 0 seconds)
```

The output tells us that three tests were executed. One of them failed. The test output also informs us of the line in which the error occured (25), and of the expected (-2) and actual (2) values. Whenever the execution of tests ends in an error, NetBeans also displays the error state visually.

While the previous tests two passed, one of them resulted in an error. Let's fix the mistake left in the Calculator class.

```
// ...
public void subtract(int number) {
    this.value -= number;
}
// ...
```

When the test are run again, they pass.

Sample output

```
Testsuite: CalculatorTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.056 sec

test-report:
test:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Unit Testing and the Parts of an Application

Unit testing tends to be extremely complicated if the whole application has been written in "Main". To make testing easier, the app should be split into small parts, each having a clear responsibility. In the previous section, we practiced this when we seperated the user interface from the application logic. Writing tests for parts of an application, such as the 'JokeManager' class from the previous section is significantly easier than writing them for program contained in "Main" in its entirety.

Test-Driven Development

Test-driven development is a software development process that's based on constructing a piece of software in small iterations. In test-driven software development, the first thing a programmer always does is write an automatically-executable test, which tests a single piece of the computer program.

The test will not pass because the functionality that satisfies the test, i.e., the part of the computer program to be examined, is missing. Once the test has been written, functionality that meets the test requirements is added to the program. The tests are then run again. If all tests pass, a new test is added, or alternatively, if the tests fail, the already-written program is corrected. If necessary, the internal structure of the program will be corrected or refactored, so that the functionality of the program remains the same, but the structure becomes clearer.

Test-driven software development consists of five steps that are repeated until the functionality of the program is complete.

1. Write a test. The programmer decides which program functionality to test and writes a test for it.
2. Run the tests and check if the tests pass. When a new test is written, the tests are run. If the test passes, the test is most likely erroneous and should be corrected - the test should only test functionality that hasn't yet been implemented.
3. Write the functionality that meets the test's requirements. The programmer implements functionality that only meets the test requirements. Note: this doesn't do things that the test does not require - functionality is only added in small increments.
4. Perform the tests. If the tests fail, there is likely to be an error in the functionality written. Correct the functionality - or, if there is no error in the functionality, fix the latest test that was performed.
5. Repair the internal structure of the program. As the size of the program increases, its internal structure is adjusted as needed. Methods that are too long are broken down into multiple parts and classes representing concepts are isolated. The tests are not modified, but are instead used to verify the correctness of the changes made to the program's internal structure - if a change in the program structure changes the functionality of the program, the tests will produce a warning and the programmer can remedy the situation.

Test-driven development

Prev Next

Programming exercise:
Exercises (2 parts)

Points
2/2

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

The exercise base contains the initial state of the previous example — it already includes the JUnit unit testing library. Follow the steps of the example, and use the principles of test-driven software development to give the wanted functionality for the exercise management program.

The exercise is in two parts:

1. Follow the steps of the example up until it's time to refactor the program and to create the class 'Exercise'. Create the classes `ExerciseManagementTest` and `ExerciseManagement`, and complete them with what the example instructs.
2. Follow the example all the way to the end. In other words, refactor the program as instructed.

Update the `partsCompleted` class method of the MainProgram to return the highest part that you have completed. You can return the exercise even if you don't complete the second part, in which case you will receive the point for the first one. Returning 2 means you have completed both.

If you desire to develop the program further (not awarded with points), you can try using test-driven development to first write a test (or a few) for removing exercises, and then implement that feature in your program. This is purely for your own amusement and is not reflected in the points in any manner!

Exercise submission instructions

How to see the solution

More about software testing

Unit testing is only a part of software testing. On top of unit testing, a developer also performs integration tests that examine the interoperability of components, such as classes, and interface tests that test the application's interface through elements provided by the interface, such as buttons.

These testing methods are covered in more detail in the more advanced courses.

You have reached the end of this section! Continue to the next section:

→ 4. Complex programs

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.