Table of Contents

Software Testing: From Theory to Practice

Welcome to **Software Testing: From Theory to Practice**!

This book covers the most important testing techniques needed to build high quality software systems. Specific topics covered are quality attributes, maintainability and testability, manual and exploratory testing, automated testing, devops, test adequacy, model-based testing, state-based testing, decision tables, reviews and inspections, design-by-contract, test-driven design, unit versus integration testing, mocks and stubs.

We expect readers to be able to:

- Create unit, integration, and system tests using current existing tools (i.e., JUnit, Mockito, and JaCoCo) that effectively test complex software systems.
- Derive test cases that deal with exceptional, corner, and bad weather cases by performing several different techniques (i.e., boundary analysis, state-based testing, decision tables) as well as able to reflect about their limitations, when and when not to apply them in a given context.
- Measure and reflect about the efficiency of the developed test suites by means of different test adequacy metrics (i.e., line, branch, condition, MC/DC coverage).
- Design and implement testable software systems by means of architectural patterns such as dependency injection, and ports and adapters.
- Write maintainable test code by avoiding well-known test code smells (e.g., Assertion Roulette, Slow or Obscure Tests).

Target audience: 1st year Computer Science students. Knowledge in programming is required. Code examples are in Java, although easy transferrable to other languages.

Authors

- Maurício Aniche, Assistant Professor in Software Engineering at Delft University of Technology

- Arie van Deursen, Full Professor in Software Engineering at Delft University of Technology

Acknowledgments:

- Annibale Panichella: chapter on mutation testing
- Wouter Polet: initial transcripts of the lectures.
- TU Delft's New Media Centre: video recording and editing

Adopting this book

This book has been used by: Delft University of Technology (Dr. Maurício Aniche), University of Zurich (Prof. Dr. Alberto Bacchelli), PUC-RS/Brazil (Prof. Dr. Bernardo Copstein).

Feel free to adopt this book in your software testing course. The slides we use in the background of our videos can be found in our video slides repository. We should release our accompanying lectures slides soon. *If you adopt this book, please let us know!*

How to contribute

We accept any kind of contribution, from typo fixes, rephrasing, illustrations, videos, exercises, code examples to full chapters. Just open a Pull Request in our GitHub repository. Or, if you feel you are not ready to contribute, but want to see some content here, just open an issue.

Note that everything you create will be licensed under the license below. We will add your name in acknowledgements.

License

Why software testing?

Welcome to the first chapter of Software Testing: From Theory to Practice! In this book, we are going to teach you how to effectively test a software and make sure it works, and how to, as much as possible, automate every step that we can.

By testing software, we mean that we are gonna explore different techniques to design test cases. For example, based on the requirements, based on the source code, analyze boundaries, and etc. And by automation, we mean making sure that a machine can do the tasks that humans would just take too much time to do, or would not be as good as a machine.

You are gonna see throughout the course that software testing is a very creative activity, and while creativity's fundamental, we are going to study systematic and rigorous ways to derive test cases. In addition, this course will be very practical. Meaning, whenever we discuss some technique, we are going to apply this technique in several examples. You should expect to see a lot of Java code throughout this course. And we hope that you can then generalize everything you learn here to test the software that you work in your real life.

And so why should we think about software testing? **Why should we actually care about software testing?** Because bugs are everywhere. And you, as a human being, probably faced a few bugs in your life. Some of them probably did not affect you that much, but definitely some other bugs really affected your life.

So just to give an impression of how easy doing mistakes is, let's start with a requirement.

Let's suppose we have this customer and this person wants a software that given a list of numbers, should return the smallest and the largest number in this list.

This looks like a very simple program to do. Let's get our hands dirty and implement it in Java:

```java
public class NumFinder {
  private int smallest = Integer.MAX_VALUE;
  private int largest = Integer.MIN_VALUE;

  public void find(int[] nums) {
    for(int n : nums) {
      if(n < smallest) smallest = n;
      else if(n > largest) largest = n;
    }
  }

  // getters for smallest and largest
}
```

The idea behind the code is as follows: we go through all the elements of this list and store the smallest number in a variable and the largest number in another variable. If n is smaller than the smallest number we have seen, we replace it.

We do the same for the largest: if n is bigger than largest, we just replace largest to n.

And, as developers, that is what we often do: we implement the requirements, and then do small checks to make sure it works. One way to do a small check would be to come up with a main method that tries the program a bit.

As an example, let's see what happens if we execute our program with some random numbers as inputs, like, 4, 25, 7, and 9.

To see if the program behaves correctly, let's just print the largest and smallest fields. Printing debug information is quite common:

```java
public class NumFinderMain {

  public static void main (String[] args) {
    NumFinder nf = new NumFinder();
    nf.find(new int[] {4, 25, 7, 9});

    System.out.println(nf.getLargest());
    System.out.println(nf.getSmallest());
  }
}
```

The output we get is: `25, 4`. This means our software works! We can ship it to our customers! **Can we...?**

We will certainly have problems with this code... Because this implementation does not work for all the possible cases. **There's a bug in there!** (Can you find the bug? Come back to the code above and look for it!)

If we pass an input like 4, 3, 2, and 1; or, in other words, numbers in decreased order, we would get an output like this: `-2147483648, 1`.

We indeed have a bug. Before anything else, let's just fix it. If we go back to the source code, we can see that this bug is actually caused by the `else if`.

```java
public class NumFinder {
  private int smallest = Integer.MAX_VALUE;
  private int largest = Integer.MIN_VALUE;

  public void find(int[] nums) {
    for(int n : nums) {
      if(n < smallest) smallest = n;

      // BUG was here!!
      if(n > largest) largest = n;
    }
  }

  // getters for smallest and largest
}
```

Take a moment here to understand why. This else if should actually just be an if.

Yes, this is indeed a very simple bug. But the point is that those mistakes can and do happen all the time. As developers, we usually deal with very complex software and it is challenging for us to have everything in our minds.

This is why we need to test software. Because bugs do happen. And they can really have a huge impact in our lifes.

And that is what we are going to focus throughout this course. Hopefully, with everything we are going to teach you, you will have enough knowledge to rigorously test your software, and make sure that bugs like thise one do not happen.

Watch our video on Youtube:
https://www.youtube.com/embed/xtLgp8LXWp8

Principles of software testing

Now that we have some basic tools to design and automate our tests, we can think more about some testing concepts. We start with some precise definitions for certain terms.

Failure, Fault and Error

We can use a lot of different words for a system that is not behaving correctly. Just to name a few we have error, mistake, defect, bug, fault and failure. As we like to be able to describe the problem in software precisely, we need to agree on a certain vocabulary. For now this comes down to three terms: **failure**, **fault**, and **error**.

A **failure** is a component of the (software) system that is not behaving as expected. An example of a failure is the well-known blue screen of death. We generally expect our pc to keep running, but it just crashes.

Failures are generally caused by faults. **Faults** are also called defects or bugs. A fault is a flaw, or mistake, in a component of the system that can cause the system to behave incorrectly. We have a fault when we have, for example, a `>` instead of `>=` in a condition.

A fault in the code does not have to cause a failure. If the code containing the fault is not being run, it can also not cause a failure. Failures only occur when the end user is using the system, when they notice it not behaving as expected.

Finally we have the **error**, also called a **mistake**. An error is a human action that cause the system to run not as expected. For example someone can forget to think about a certain corner case that the code might run into. This then creates a fault in the code, which can result in a failure.

Watch our video on Youtube: https://www.youtube.com/embed/zAty8Rpg92I

Verification and Validation

We keep extending our vocabulary a bit with two more concepts. Here we introduce **verification** and **validation**. Both concepts are about assessing the quality of a system and can be described by a single question.

Validation: Are we building the right software? Validation concerns the features that our system offers and the customer, for who the system is made. Is the system that we are building actually the system that the users want? Is the system actually useful?

Verification on the other hand is about the system behaving as it is supposed to according to the specification. This mostly means that the systems behaves without any bugs, like it is said it should behave. This does not guarantee that the system is useful. That is a matter of validation. We can summarize verification with the question: **Are we building the system right?**

In this course, we mostly focus on verification. However, validation is also very important when it comes to building successful software systems.

Watch our video on Youtube:
https://www.youtube.com/embed/LZ3Fb2Jq7yw

Tests, tests and more tests

If we want to test our systems well, we just keep adding more tests until it is enough, right? Actually a very important part of software testing is knowing when to stop testing. Creating too few tests will leave us with a system that might not behave as intended. On the other hand, creating test after test without stopping will probably cost too much time to create and at some point will make the whole test suite too slow. A couple of tests are executed in a very short amount of time, but if the amount of tests becomes very high the time needed to execute them will naturally increase as well. We discuss a couple of important concepts surrounding this subject.

First, **exhaustive testing** is impossible. We simply cannot test every single case a method might be executing. This means that we have to prioritize the tests that we do run.

When prioritizing the test cases that we make, it is important to notice that **bugs are not uniformly distributed**. If we have some components in our system, then they will not all have the same amount of bugs. Some components probably have more bugs than others. We should think about which components we want to test most when prioritizing the tests.

A crucial notion for software testing is that **testing shows the presence of defects; however, testing does not show the absence of defects**. So while we might find more bugs by testing more, we will never be able to say that our software is 100% bug-free, because of the tests.

To test our software we need a lot of variation in our tests. When testing a method we want variety in the inputs, for example, like we saw in the examples above. To test the software well, however, we also need variation in the testing strategies that we apply. This is described by the **pesticide paradox**: "Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual." There is no testing strategy that guarantees that the tested software is bug-free. So, when using the same strategy all the time, we will miss some of the defects that are in the software. This is the residue that is talked about in the pesticide paradox. From the pesticide paradox we learn that we have to use different testing strategies on the same software to minimize the bugs left in the software. When learning the various testing strategies in this reader, keep in mind that you want to combine them when you are testing your software.

Combining these testing strategies is a great idea, but it can be quite challenging. For example, testing a mobile app is very different from testing a web application or a rocket. In other words: **testing is context-dependent**. The way that we test depends on the context of the software that we test.

Finally, while we are mostly focusing on verification when we create tests, we should not forget that just having a low amount of bugs is not enough. If we have a program that works like it is specified, but is of no use for its users, we still do not have good software. This is called the **absence-of-errors fallacy**. We cannot forget about the validation part, where we check if the software meets the users' needs.

Watch our video on Youtube: https://www.youtube.com/embed/dkbvb_wTN-M

Exercises

Exercise 1. Having a certain terminology helps testers to explain the problems they have with a program or in their software.

Below is a small conversation. Fill each of the caps with: failure, fault, or error.

- **Mark**: Hey, Jane, I just observed a (1) in our software: if the user has multiple surnames, our software doesn't allow them to sign in.
- **Jane**: Oh, that's awful. Let me debug the code so that I can find the (2) . *(a few minutes later)*
- **Jane**: Mark, I found it! It was my (3) . I programmed that part, but never thought of this case.
- **Mark**: No worries, Jane! Thanks for fixing it!

Exercise 2. Kelly, a very experienced software tester, visits Books!, a social network focused on matching people based on books they read. Users do not report bugs so often; Books! developers have strong testing practices in place. However, users do say that the software is not really delivering what it promises.

What testing principle applies to this problem?

Exercise 3. Suzanne, a junior software testing, just joined a very large online payment company in the Netherlands. As a first task, Suzanne analyzed their past two years of bug reports. Suzanne observes that more than 50% of bugs have been happening in the 'International payments module.

Suzanne then promises her manager that she will design test cases that will completely cover the 'International payments' module, and thus, find all the bugs.

Which of the following testing principles might explain why this is **not** possible?

1. Pesticide paradox.
2. Exhaustive testing.
3. Test early.
4. Defect clustering.

Exercise 4. John strongly believes in unit testing. In fact, this is the only type of testing he actually does at any project he's in. All the testing principles below, but one, might help in convincing John that he should also focus on different types of testing.

Which of the following **is the least related** related to help John in moving away from his 'only unit testing' approach?

1. Pesticide paradox.
2. Tests are context-dependent.
3. Absence-of-errors fallacy.
4. Test early.

References

- Chapters 1-3 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.

Software testing automation (with JUnit)

Before we explore different testing techniques, let's first get used to software testing automatin frameworks, like JUnit. For now, let's just use our experience as software developers to devise test cases.

The Roman Numeral problem

It is our goal to implement a program that receives a string as a parameter containing a roman number and then converts it to an integer.

In roman numeral, letters represent values:

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

We can combine these letters to form numbers. The letters should be ordered from the highest to the lowest value. For example `CCXVI` would be 216.

When we put a lower value in front of a higher one, we substract that value from the higher value. For example we make 40 not by XXXX, but instead we use $50 - 10 = 40$ and have the roman number `XL`. Combining both these principles we could give our method `MDCCCXLII` and it should return 1842.

Watch our video on Youtube:
https://www.youtube.com/embed/srJ91NRpT_w

A possible implementation for this Roman Numeral is as follows:

```java
public class RomanNumeral {
  private static Map<Character, Integer> map;

  static {
    map = new HashMap<>();
    map.put('I', 1);
    map.put('V', 5);
    map.put('X', 10);
    map.put('L', 50);
    map.put('C', 100);
    map.put('D', 500);
    map.put('M', 1000);
  }

  public int convert(String s) {
    int convertedNumber = 0;

    for (int i = 0; i < s.length(); i++) {
      int currentNumber = map.get(s.charAt(i));
      int next = i + 1 < s.length() ? map.get(s.charAt(i + 1)) : 0;

      if (currentNumber >= next) {
        convertedNumber += currentNumber;
      } else {
        convertedNumber -= currentNumber;
      }
    }

    return convertedNumber;
  }
}
```

Now we have to think about tests for this method. Use your experience as a developer to get as many test cases as you can. To get you started, a few examples:

- Just one letter (C = 100)
- Different letters combined (CLV = 155)
- Subtractive notation (CM = 900)

Let's now automate these manually devised test cases.

The JUnit Framework

Testing frameworks enables us to write our test cases in a way that they can be easily executed by the machine. For Java, the standard framework to write automated tests is JUnit (the frameworks for the languages work similarly), and its most recent version is 5.x.

To automate the tests we need a Java class. This is usually named the name of the class under test, followed by "Test". To create an automated test, we make a method with the return type `void`. For the execution the name of the method does not matter, but we always use it to describe the test. To make sure that JUnit considers the method to be a test, we use the `@Test` annotation. To use this annotation you have to import `org.junit.jupiter.api.Test`.

In the method itself we execute the code that we want to test. After we have done that, we check is the result is what we would expect. To check this result, we use assertions.

A couple of useful assertions are:

- `assertEquals(expected, actual)` : Passes if the expected and actual values are equal, fails otherwise. Be sure to pass the expected value as the first argument and the actual value (the value that, for example, the method returns) as second argument. Otherwise the fail message of the test will not make sense.
- `assertTrue(condition)` : Passes if the condition evaluates to true, fails otherwise.
- `assertFalse(condition)` : Passes if the condition evaluates to false, fails otherwise.

More assertions and additional arguments can be found in JUnit's documentation. To make easy use of the assertions and to import them all in one go, you can use `import static org.junit.jupiter.api.Assertions.*;` .

We need a class and methods annotated with `@Test` to automate our test cases. The three cases we had, can be automated as follows.

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class RomanNumeralTest {

  @Test
  void convertSingleDigit() {
    RomanNumeral roman = new RomanNumeral();
    int result = roman.convert("C");

    assertEquals(100, result);
  }

  @Test
  void convertNumberWithDifferentDigits() {
    RomanNumeral roman = new RomanNumeral();
    int result = roman.convert("CCXVI");

    assertEquals(216, result);
  }

  @Test
  void convertNumberWithSubtractiveNotation() {
    RomanNumeral roman = new RomanNumeral();
    int result = roman.convert("XL");

    assertEquals(40, result);
  }
}
```

Notice that we first create an instance of `RomanNumeral` . Then we execute or run the `convert` method, which is the method we want to test. Finally we assert that the result is what we would expect.

Do you see more test cases? Go ahead and implement them!

Watch our video on Youtube: https://www.youtube.com/embed/XS4-93Q4Zy8

The need for systematic software testing

We devised three test cases in our exemple above. How did we do it? We used our experience as software developers.

However, although our experience indeed helps us deeply in finding bugs, this might not be enough:

- It is highly prone to mistakes. Maybe you might forget one test case.
- It varies from person to person. We want any developer in the world to be able to test software.
- It is hard to know when to stop, based only on our gut feelings.

This is why, throughout the course, we will study more systematic techniques to test software.

Watch our video on Youtube: https://www.youtube.com/embed/xyV5fZsUH9s

An introduction to test code quality

In practice, developers write (and maintain!) thousands of test code lines. Taking care of the quality of test code is therefore of utmost importance. Whenever possible, we'll introduce you to some best practices in test code engineering.

For example, in the example above, we create the `roman` object four times. This is good, because we want to start fresh with each test. If the method would in some way change the object, we do not want this to carry over to another test. However, now we have the same line of code in each test method. The risk of this code duplication is that we might do it wrong in one of the cases, or if we want to change the code somewhere, we forget to change it everywhere.

We can write everything we want to do at the start of each test in just one method. In order to do so, we will use the `@BeforeEach` annotation. Like the tests, we annotate a method with this annotation. Then JUnit knows that before it runs the code in a test method is has to run the code in the `BeforeEach` method.

In the test code we just wrote, we can instantiate the `roman` object inside a method annotated with `BeforeEach`. In this case it is just one line that we move out of the test methods, but as your tests become more complicated this approach becomes more important.

The new test code would look as follows.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class RomanNumeralTest {

  private RomanNumeral roman;

  @BeforeEach
  void setup() {
    roman = new RomanNumeral();
  }

  @Test
  void convertSingleDigit() {
    roman = new RomanNumeral();
    int result = roman.convert("C");

    assertEquals(100, result);
  }

  @Test
  void convertNumberWithDifferentDigits() {
    roman = new RomanNumeral();
    int result = roman.convert("CCXVI");

    assertEquals(216, result);
  }

  @Test
  void convertNumberWithSubtractiveNotation() {
    roman = new RomanNumeral();
    int result = roman.convert("XL");

    assertEquals(40, result);
  }
}
```

We will discuss test code quality in a more systematic way in a future chapter.

Test design vs test execution

In practice, we usually have two distinct tasks when performing software testing.

The first one is about analysing and designing test cases, where the goal is for us to think about everything we wanna test so that we are sure that our software works as expected. Usually, this phase is done by humans, although we will explore the state-of-the-art in software testing research, where machines also try to devise test cases for us. We are going to explore different strategies that a person can do to design good test cases, such as functional testing, boundary testing, and structural testing.

The second part is about executing the tests we created. And we often do it by actually running the real software or by exercising its classes, and making sure that the software does what it is supposed to do. Although this phase can also be done by humans, this is an activity that we can easily automate. Meaning, we can write a program that run our software and executes the test cases. Writing these test programs or, as we call, writing automated tests, is also a fundamental part of this course.

To sum up, you should do two activities when testing your software. The first one, the more human part, which is to think and design test cases in the best way possible. And the second phase, which is to execute the tests against the software under test, and make sure that it behaves correctly. And that, we will always try to automate it as much as possible.

Side note 1: If you're very interested on understanding why it is so hard to teach machines to design test cases for us, and therefore, remove the human out of the loop, you can read this amazing paper called "The Oracle Problem in Software Testing: A Survey".

Side note 2: In industry, the term *automated software testing* is related to the execution of test cases; in other words, JUnit code. In academia, whenever a research paper says *automated software testing*, it means automatically designing test cases (and not only the JUnit code).

Watch our video on Youtube:
https://www.youtube.com/embed/pPv37kPqvAE

Exercises

Exercise 1. Implement the `RomanNumeral` class. Then, write as many tests as you can for it, using JUnit.

For now, do not worry about how to derive test cases. Just follow your intuition.

Exercise 2. Choose a problem from Codebat. Solve it. Then, write as many tests as you can for it, using JUnit.

For now, do not worry about how to derive test cases. Just follow your intuition.

References

- Pragmatic Unit Testing in Java 8 with Junit. Langr, Hunt, and Thomas. Pragmatic Programmers, 2015.
- Barr, Earl T., Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey." IEEE transactions on software engineering 41, no. 5 (2014): 507-525. Harvard

Specification-Based Testing

In this chapter, we explore **specification-based testing** techniques. In such techniques, we use the *requirements* of the program as an input for our testing.

In simple words, given a requirement, we will aim at devising a set of inputs, each tackling one part (or partition, as we will call later) of the program.

Given that specification-based techniques require no knowledge of how the software inside the "box" is structured (i.e., we do not care if it's developed in Java or Python), they are also known as black box testing.

Partitioning the input space

Programs are usually too complex to be tested with just a single test. There are different cases in which the program is executed and its execution often depends on various factors, such as the input you pass to the program.

Let's use a small program as an example. The specification below talks about a program that decides whether an year is leap or not.

> **Requirement: Leap year**
>
> Given an year as an input, the program should return true if the the provided year is leap; false if it is not.
>
> A year is a leap year if:
>
> - the year is divisible by 4
> - the year is not divisible by 100
> - exceptions are years that are divisible by 400, which are also leap years.

To find a good set of tests, often referred to as a *test suite*, we break up (or to part, or to divide) the testing of a program in classes. Moreover, we have divide the input space of the program in such a way that each class is 1) disjoint, i.e., represents a unique behavior of the program; in other words, no two partitions represent the same behavior, 2) can easily verify whether that behavior is correct or not.

Classes for the leap year problem

By looking at the requirements, we can derive the following classes/partitions:

- Year is divisible by 4, but not divisible by 100 = leap year, TRUE
- Year is divisible by 4, divisible by 100, divisible by 400 = leap year, TRUE
- Not divisible by 4 = not a leap year, FALSE
- Divisible by 4, divisible by 100, but not divisible by 400 = not leap year, FALSE

Note how each class above exercises the program in different ways.

TODO: Add a picture here showing the input domain and its classes.

Watch our video on Youtube:
https://www.youtube.com/embed/kSLbxmXcPPI

Equivalence partitioning

Now that we just divided the input space into different classes, we can automate it. As we discussed before, the test design is still a human activity, which we just did. Now, we should automate the execution of test.

The partitions, as you see above, are not test cases we can directly implement. Each partition might be instantiated by an infinite number of inputs. For example, the partition *year not divisible by 4*, we have an infinite number of numbers that are not divisible by 4 that we could use as concrete inputs to the program. It is just impractical to test them all. Then how do we know which concrete input to instantiate for each of the partitions?

As we discussed earlier, each partition exercises the program in a certain way. In other words, all input values from one specific partition will make the program behave in the same way. Therefore, any input we select should give us the same result. And we assume that, if the program behaves correctly for one given input, it will work correctly for all other inputs from that class. This idea of inputs being equivalent to each other is what we call **equivalence partitioning**.

It then does not matter which precise input we pick. And picking one test case per partition will be enough.

Let's now write some JUnit tests. Remember that the name of a test method in JUnit can be anything? It is a good practice to name your test method after the partition that the method tests.

Example: The Leap Year specification has been implemented by a developer in the following way:

```java
public class LeapYear {

  public boolean isLeapYear(int year) {
    if (year % 400 == 0)
      return true;
    if (year % 100 == 0)
      return false;

    return year % 4 == 0;
  }
}
```

Given the classes we devised before, we know we have 4 test cases in total. We can choose any input in a certain partition. We will use the following inputs for each partition:

- 2016, divisible by 4, not divisible by 100.
- 2000, divisible by 4, also divisible by 100 and by 400.
- 39, not divisible by 4.
- 1900, divisible by 4 and 100, not by 400.

Implementing this using JUnit gives the following code for the tests.

```java
package tudelft.leapyear;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class LeapYearTests {

  private LeapYear leapYear;

  @BeforeEach
  public void setup() {
    leapYear = new LeapYear();
  }

  @Test
  public void leapYearsNotCenturialTest() {
    boolean leap = leapYear.isLeapYear(2016);

    assertTrue(leap);
  }

  @Test
  public void leapYearsCenturialTest() {
    boolean leap = leapYear.isLeapYear(2000);

    assertTrue(leap);
  }

  @Test
  public void nonLeapYearsTest() {
    boolean leap = leapYear.isLeapYear(39);

    assertFalse(leap);
  }

  @Test
  public void nonLeapYearsCenturialTest() {
    boolean leap = leapYear.isLeapYear(1900);

    assertFalse(leap);
  }
}
```

Note that each test method covers one of the partitions and the naming of the method refers to the partition it covers.

Some help if you are still learning JUnit: The `setup` method is executed before each test, thanks to the `BeforeEach` annotation. It creates the `LeapYear`. This is then used by the tests to execute the method under test. In each test we first determine the result of the method. After the method returns a value we assert that this is the expected value.

Watch our video on Youtube: https://www.youtube.com/embed/mXmFiiifwaE

Category-Partition Method

So far we derived partitions by just looking at the specification of the program. We basically used our experience and knowledge to derive the test cases. We now go over a more systematic way of deriving these partitions: the **Category-Partition** method.

The method gives us 1) a systematic way of deriving test cases, based on the characteristics of the input parameters, 2) minimize the amount of tests to a feasible amount.

We first go over the steps of this method and then we illustrate the process with an example.

1. Identify the parameters, or the input of the program. For example, the parameters your classes and methods receive.
2. Derive characteristics of each parameter. For example, an `int year` should be a positive integer number between 0 and infinite.
    - Some of these characteristics can be found directly in the specification of the program.
    - Others cannot be found from specifications. For example an input cannot be `null` if the method does not handle that well.
3. Add constraints, as to minimize the test suite.
    - Identify invalid combinations. For example, some characterics might not be able to be mixed with other characteristics.
    - Exceptional behavior does not always have to be combined with all the different values of the other inputs. For example, trying a single `null` input might be enough to test that corner case.
4. Generate combinations of the input values. These are the test cases.

> **Requirement: Christmas discount**
>
> The system should give 25% discount on the raw amount of the cart when it is Christmas. The method has two input parameters: the total price of the products in the cart and the date. When it is not Christmas it just returns the original price, otherwise it applies the discount.

Now following the category partition method:

1. We have two parameters:
    - The current date
    - The total price
2. Now for each parameter we define the characteristics:

    - Based on the requirements, the only important characteristic is that the date can be either Christmas or not.
    - The price can be a positive number, or maybe 0 for some reason. Technically the price can also be a negative number. This is an exceptional case, as you cannot really pay a negative amount.
3. The amount of characteristics and parameters is not too high in this case. Still we know that the negative price is an exceptional case. Therefore we can test that with just one combination instead of with both a date that is Christmas and not Christmas.

4. We combine the other characteristics to get the test cases. These are the following:

    - Positive price on Christmas
    - Positive price not on Christmas
    - Price of 0 on Christmas
    - Price of 0 not on Christmas
    - Negative price on Christmas

Now we can implement these test cases. Each of the test cases corresponds to one of the partitions that we want to test.

Watch our video on Youtube: https://www.youtube.com/embed/frzRmafsPBk

One more example of specification-based testing

Imagine the following requirement:

> **Chocolate bars**
>
> A package should store a total number of kilos. There are small bars (1 kilo each) and big bars (5 kilos each). We should calculate the number of small bars to use, assuming we always use big bars before small bars. Return -1 if it can't be done.
>
> The input of the program is thus the number of small bars, the number of big bars, and the total amount of kilos to store.

In this example, the partitions are a bit more "hidden". We have to really understand the problem in order to derive the partitions. You should spend some time (try to even implement it!!) in understanding it.

Now, let's think about the classes/partitions:

- **Need only small bars**. A solution that only uses the provided small bars.
- **Need only big bars**. A solution that only uses the provided big bars.
- **Need Small + big bars**. A solution that has to use both small and big bars.
- **Not enough bars**. A case in which it's not possible, because there are not enough bars.

- **Not from the specs**: An exceptional case.

For each of these classes, we can devise concrete test cases:

- **Need only small bars**. small = 4, big = 2, total = 3
- **Need only big bars**. small = 5, big = 3, total = 10
- **Need Small + big bars**. small = 5, big = 3, total = 17
- **Not enough bars**. small = 1, big = 1, total = 10
- **Not from the specs**: small = 4, big = 2, total = -1

This example shows why deriving good test cases is challenging. Specifications can be complex and we need to fully understand the problem!

Watch our video on Youtube:
https://www.youtube.com/embed/T8caAUwgquQ

Random testing vs specification-based testing

One might think: but what if, instead of looking at the requirements, a tester just keeps giving random inputs to the program? **Random testing** is indeed a popular black-box technique where programs are tested by generating random inputs.

Although random testing can definitely help us in finding bugs, it is not an effective way to find bugs in a large input space. Human testers use their experience and knowledge of the program to test trouble-prone areas more effectively. However, where humans can generate few tests in a short time period such as a day, computers can generate millions. A combination of random testing and partition testing is therefore the most beneficial.

In future chapters, we'll discuss fuzzing test and AI-based testing. There, you will learn more about automated random testing.

Exercises

Exercise 1. What is an Equivalence Partition?

1. A group of results that is produced by one method
2. A group of results that is produced by one input into different methods
3. A group of inputs that all make a method behave the same way
4. A group of inputs that exactly gives the same output in every method

Exercise 2. We have a program called FizzBuzz. It does the following: Given an int n, return the string form of the number followed by "!". So the int 6 yields "6!". Except if the number is divisible by 3 use "Fizz" instead of the number, and if the number is divisible by 5 use "Buzz", and if divisible by both 3 and 5, use "FizzBuzz".

A novice tester is trying hard to devise as many tests as she can for the FizzBuzz method. She came up with the following tests:

- T1 = 15
- T2 = 30
- T3 = 8
- T4 = 6
- T5 = 25

Which of these tests can be removed while keeping a good test suite?

What concept can we use to determine the tests that can be removed?

Exercise 3. See a slightly modified version of HashMap's `put` method Javadoc. (Source code here).

```
/**
 * Puts the supplied value into the Map,
 * mapped by the supplied key.
 * If the key is already on the map, its
 * value will be replaced by the new value.
 *
 * NOTE: Nulls are not accepted as keys;
 *  a RuntimeException is thrown when key is null.
 *
 * @param key the key used to locate the value
 * @param value the value to be stored in the HashMap
 * @return the prior mapping of the key, or null if there was none.
 */
public V put(K key, V value) {
  // implementation here
}
```

Apply the category/partition method. What are the minimal and most suitable partitions?

Exercise 4. Zip codes in country X are always composed of 4 numbers + 2 letters, e.g., 2628CD. Numbers are in the range [1000, 4000]. Letters are in the range [C, M].

Consider a program that receives two inputs: an integer (for the 4 numbers) and a string (for the 2 letters), and returns true (valid zip code) or false (invalid zip code).

A tester comes up with the following partitions:

1. [0,999]
2. [1000, 4000]
3. [2001, 3500]
4. [3500, 3999]
5. [4001, 9999]
6. [A-C]
7. [C-M]
8. [N-Z]

Note that with [a, b] all numbers between and including a and b are in the domain. The same goes with letters like [A-Z].

Which of these partitions are valid (and good) partitions, i.e. which can actually be used as partitions? Name each of the valid partitions, corresponding to how they exercise the program.

Exercise 5. See a slightly modified version of HashSet's `add()` 's Javadoc below. Apply the category/partition method. What is **the minimal and most suitable partitions** for the `e` input parameter?

```
/**
 * Adds the specified element to this set if it
 * is not already present.
 * If this set already contains the element,
 * the call leaves the set unchanged
 * and returns false.
 *
 * If the specified element is NULL, the call leaves the
 * set unchanged and returns false.
 *
 * @param e element to be added to this set
 * @return true if this set did not already contain
 *   the specified element
 */
public boolean add(E e) {
    // implementation here
}
```

Exercise 6. Which of the following statements **is false** about applying the category/partition method in method below?

```
/**
 * Puts the supplied value into the Map,
 * mapped by the supplied key.
 * If the key is already on the map, its
 * value will be replaced by the new value.
 *
 * NOTE: Nulls are not accepted as keys;
 *  a RuntimeException is thrown when key is null.
 *
 * @param key the key used to locate the value
 * @param value the value to be stored in the HashMap
 * @return the prior mapping of the key,
 *  or null if there was none.
 */
public V put(K key, V value) {
  // implementation here
}
```

1. The specification does not specify any details about the `value` input parameter, and thus, experience should be used to partition it, e.g., `value` being null and not null.
2. The number of tests generated by the category/partition method can grow quickly, as the chosen partitions for each category are later combined one-by-one. This is not a practical problem to the `put()` method because the number of categories and their partitions is small.

3. In an object-oriented language, besides using the method's input parameters to explore partitions, we should also consider the internal state of the object (i.e., the class's attributes), as it can also affect the behavior of the method.

4. With the information in hands, it is not possible to perform the category/partition method, as the source code is required for the last step of the category/partition method: adding constraints.

Exercise 7. Suppose a `find` program that finds occurrences of a pattern in a file. The program has the following syntax:

```
find <pattern> <file>
```

A tester, after reading the specs and following the Category-Partition method, devised the following test specification:

- Pattern size: empty, single character, many characters, longer than any line in the file.
- Quotting: pattern is quoted, pattern is not quoted, pattern is improperly quoted.
- File name: good file name, no file name with this name, omitted.
- Occurrences in the file: none, exactly one, more than one.
- Occurrences in a single line, assuming line contains the pattern: one, more than one.

However, the number of combinations is now too high. What actions could we take to reduce the number of combinations?

References

- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Chapter 10 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), 676-686.

Boundary testing

A high number of bugs happen in the **boundaries** of your program. In this chapter we are going to derive tests for these boundaries.

The boundaries can reside between partitions or specific conditions. Off-by-one errors, i.e., when your program outcome is "off by one", often occur because of the lack of boundary testing.

Boundaries in between classes/partitions

Whenever we devised partitions/classes, these classes have "close boundaries" with the other classes.

We can find such boundaries by finding a pair of consecutive input values $[p_1, p_2]$, where p_1 belongs to partition A, and p_2 belongs to partition B. In other words, the boundary itself is where our program changes from our class to the other. As testers, we should make sure that everything works smoothly (i.e., the program still behaves correctly) near these values.

> **Requirement: Calculating the amount of points of the player**
>
> Our program has two inputs: the score of the player and the remaining life of the player. If the player's score is below 50, then it always adds 50 points. If the remaining life is above or equal to 3 lives and the score is greater than or equals to 50, the player's score is doubled.

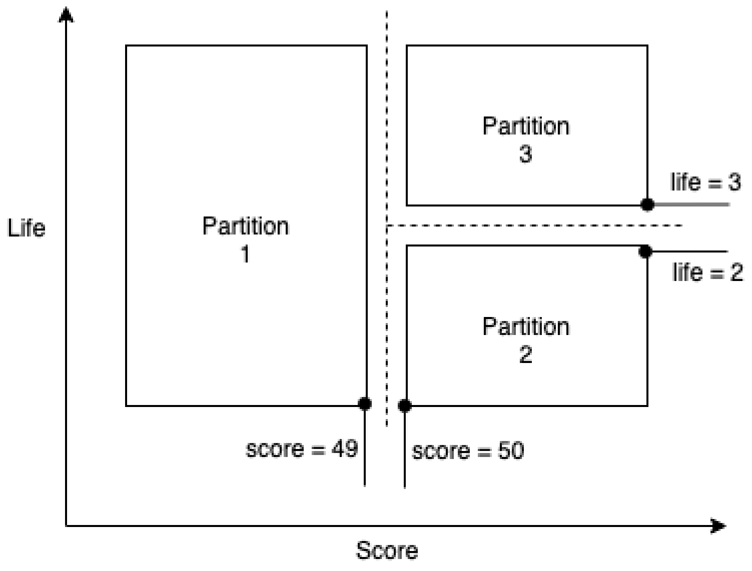
When devising the partitions to test this method, we come up with the following partitions:

1. Score < 50
2. Score >= 50 and remaining life < 3
3. Score >= 50 and remaining life >= 3

When the score is strictly smaller than 50 it is part of the first partition. From a score of 50 the test case will be part of partitions 2 or 3. Now we have found a boundary between partitions 1 and 2, 3. This boundary is between the score of 49 and 50.

We also find a boundary between partitions 2 and 3. This boundary is between the remaining life. When the remaining life is smaller than 3, it belongs to partition 2; otherwise it belongs to partition 3.

We can visualize these partitions with their boundaries in a diagram.



To sum up: you should devise tests for the inputs at the boundaries of your classes.

Analysing conditions

We briefly discussed the off-by-one errors earlier. Errors where the system behaves incorrectly for values on and close to the boundary are very easily made. In practice, think of how many times the bug was in the boolean condition of your `if` or `for` condition, and the fix was basically replacing a `>=` by a `>`.

When we have a properly specific condition, e.g., `x > 100`, we can analyse the boundaries of this condition. First, we need to go over some terminology:

- On-point: the value that is on the boundary. This is the value we see in the condition.
- Off-point: the value closest to the boundary that flips the conditions. So if the on-point makes the condition true, the off point makes it false and vice versa. Note: when dealing with equalities or non equalities (e.g. $x == 6$ or $x! = 6$), there are two off-points; one in each direction.
- In-points are all the values that make the condition true.
- Out-points are all the values that make the condition false.

Note that, depending on the condition, an on-point can be either an in- or an out-point.

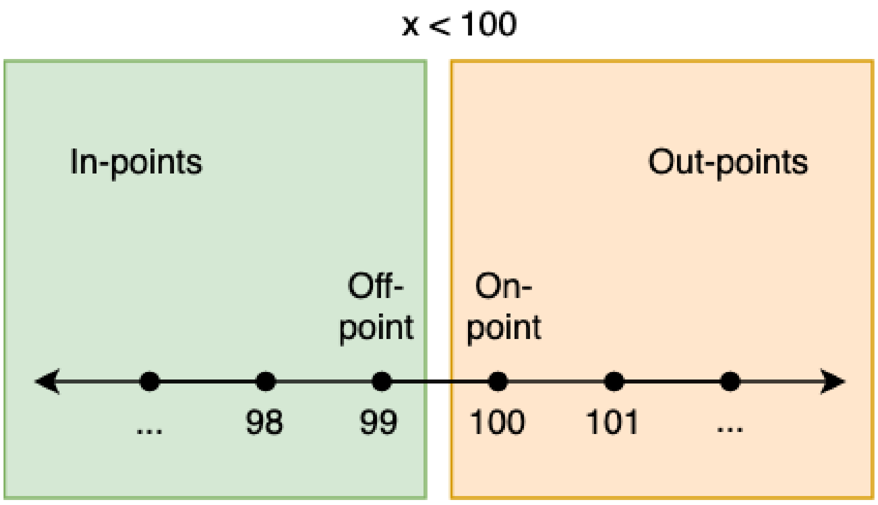
Example: Suppose we have a program that adds shipping costs when the total price is below 100.

The condition used in the program is $x < 100$.

- The on-point is 100, as that is the value in the condition.
- The on-point makes the condition false, so the off-point should be the closest number that makes the condition true. This will be 99, $99 < 100$ is true.
- The in-points are all the values smaller than or equal to 99.

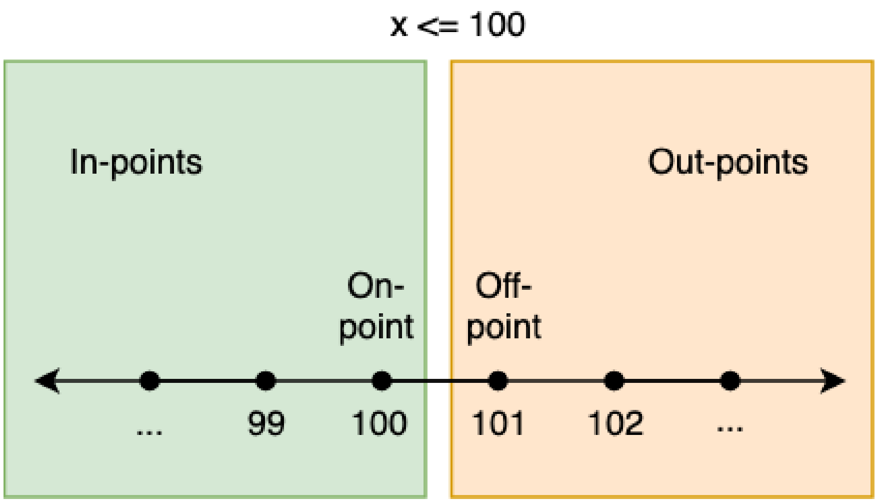
- The out-points are all values larger than or equal to 100.

We show all these points in the diagram below.



Now, let's compare it to the next condition $x <= 100$ (note how similar they are; the only difference is that, in this one, we use smaller than or equals to):

- The on-point is still 100: this is the point in the condition
- Now the condition is true for the on-point. So, the off-point should make the condition false; the off-point is 101.



Note that, in the diagram, the on-point is part of the in-points, and the off-point is part of the out-points.

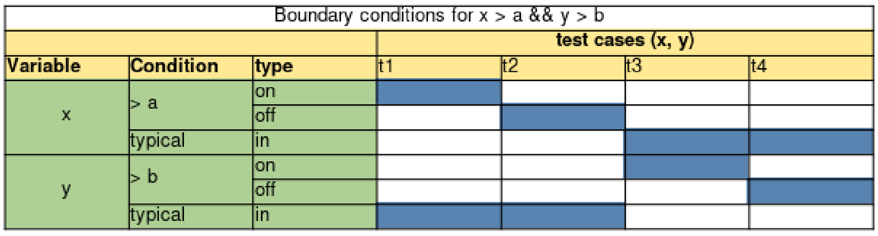
As a tester, you devise test cases for these different points.

Now that we know the meaning of these different points we can start the boundary analysis in more complicated cases. In the previous example, we looked at one condition and its boundary. However, in most programs you will find statements that consist of multiple conditions.

To test the boundaries in these more complicated decisions, we use the **simplified domain testing strategy**. The idea of this strategy is to test each boundary separately, i.e. independent of the other conditions. To do so, for each boundary:

- We pick the on- and off-point and we create one test case each.
- If we use multiple variables, we need values for those as well. As we want to test each boundary independently, we choose in-points for the other variables. Note: We always choose in points, regardless of the two boolean expressions being connected by ANDs or ORs. In practice, we want all the other conditions to return true, so that we can evaluate the outcome of the condition under test independently.
- It is important to vary these in-points and to not choose the on- or off-point. This gives us the ability to partially check that the program gives the correct results for some in-points. If we would set the in-point to the on- or off-point, we would be testing two boundaries at once.

To find these values and display the test cases in a structured manner, we use a **domain matrix**. In general the table looks like the following:



Boundary conditions for x > a && y > b

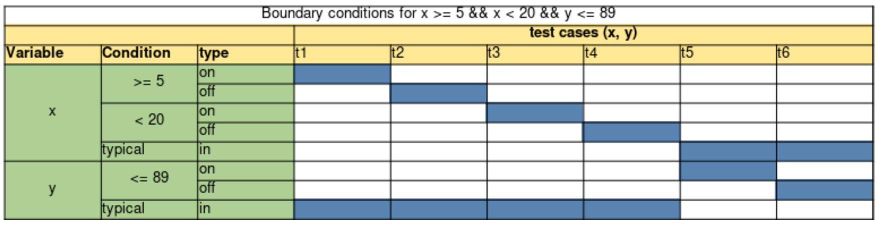
			test cases (x, y)			
Variable	**Condition**	**type**	t1	t2	t3	t4
x	> a	on	🟦			
		off		🟦		
	typical	in			🟦	🟦
y	> b	on			🟦	
		off				🟦
	typical	in	🟦	🟦		

In this template, we have two conditions with two parameters (see the $x > a \land y > b$ condition). We list the variables, with all their conditions. Then each condition has two rows: one for the on-point and one for the off-point. Each variable has an additional row for the typical (or in-) values. These are used when testing the other boundary.

Each column that corresponds to a test case has two colored cells. In the colored cells you have to fill in the correct values. Each of these pairs of values will then give a test case. If we implement all the test cases that the domain matrix gives us, we exerise each boundary both for the on- and off-point independent of the other parameters.

Example. We have the following condition that we want to test: `x >= 5 && x < 20 && y <= 89`

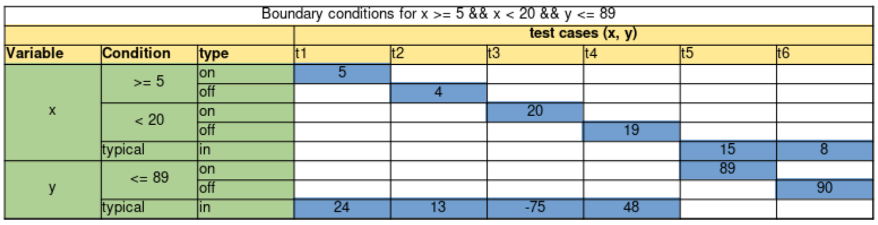
We start by making the domain matrix, having space for each of the conditions and both parameters.



Boundary conditions for x >= 5 && x < 20 && y <= 89

			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	>= 5	on	🟦					
		off		🟦				
	< 20	on			🟦			
		off				🟦		
	typical	in					🟦	🟦
y	<= 89	on					🟦	
		off						🟦
	typical	in	🟦	🟦	🟦	🟦		

Here you see that we need 6 test cases, because there are 3 conditions. Now it is time to fill in the table. We get the on- and off-points like in the previous example.



Boundary conditions for x >= 5 && x < 20 && y <= 89

			test cases (x, y)					
Variable	Condition	type	t1	t2	t3	t4	t5	t6
x	>= 5	on	5					
		off		4				
	< 20	on			20			
		off				19		
	typical	in					15	8
y	<= 89	on					89	
		off						90
	typical	in	24	13	-75	48		

Now we have derived the six test cases that we can use to test the boundaries.

Watch our video on Youtube:
https://www.youtube.com/embed/rPcMJg62wM4

Boundaries that are not so explicit

Let's revisit the example from the specification-based techniques chapter. There, we had a program where the goal was to return the amount of bars needed in order to build some boxes of chocolates:

> **Chocolate bars**
>
> A package should store a total number of kilos. There are small bars (1 kilo each) and big bars (5 kilos each). We should calculate the number of small bars to use, assuming we always use big bars before small bars. Return -1 if it can't be done.
>
> The input of the program is thus the number of small bars, the number of big bars, and the total amount of kilos to store.

And these were the classes we derived after applying the category/partition method:

- **Need only small bars**. A solution that only uses the provided small bars.
- **Need only big bars**. A solution that only uses the provided big bars.
- **Need Small + big bars**. A solution that has to use both small and big bars.
- **Not enough bars**. A case in which it's not possible, because there are not enough bars.
- **Not from the specs**: An exceptional case.

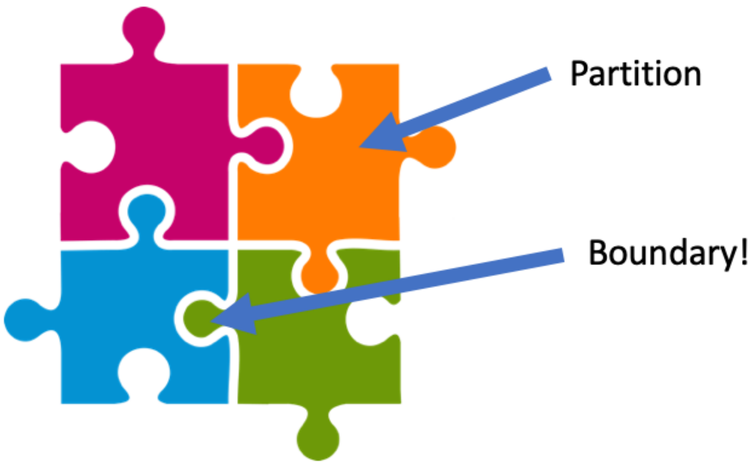
A developer implemented the following code for the requirement, and all the tests pass.

```java
public int calculate(int small, int big, int total) {
  int maxBigBoxes = total / 5;
  int bigBoxesWeCanUse = maxBigBoxes < big ? maxBigBoxes : big;

  total -= (bigBoxesWeCanUse * 5);

  if(small <= total)
      return -1;
  return total;
}
```

However, another developer tried `(2,3,17)` as an input and the program crashed. After some debugging, they noticed that the if statement should had been `if(small < total)` instead of `if(small <= total)`. This smells like a bug that could had been caught via boundary testing!

Note that the test `(2,3,17)` belongs to the **need small + big bars** partition. In this case, the program will make use of all the big bars (there are 3 available) and then *all* the small bars available (there are 2 available). Note that the buggy program would work if we had 3 available small bars (having `(3, 3, 17)` as input). This is a boundary.

How can we apply boundary testing here?



Boundaries also happen when we are going from "one partition" to another. In these cases, what we should do is to devise test cases for a sequence of inputs that move from one partition to another.

For example, let's focus on the bug caused by the `(2,3,17)` input.

- `(1,3,17)` should return *not possible* (1 small bar is not enough). This test case belongs to the **not enough bars** partition.
- `(2,3,17)` should return 2. This test case belongs to **need for small + big bars** partition.

There is a boundary between the `(1,3,17)` and the `(2,3,17)`. We should make sure the software still behaves correctly in these cases.

Let's explore another boundary. Let's focus on the **only big bars** partition. We should find inputs that transition from this partition to another one:

- `(10, 1, 10)` returns 5. This input belongs to the **need small + big bars** partition.
- `(10, 2, 10)` returns 0. This input belongs to the **need only big bars** partition.

One more? Let's focus on the **only small bars** partition:

- `(3, 2, 3)` returns 3. We need only small bars here, and therefore, this input belongs to the **only small bars** partition.
- `(2, 2, 3)` returns -1. We can't make the boxes. This input belongs to the **Not enough bars** partition.

A partition might make boundaries with other partitions. See:

- `(4, 2, 4)` returns 4. We need only small bars here, and therefore, this input belongs to the **only small bars** partition.
- `(4, 2, 5)` returns 0. We need only bigs bars here, and therefore, this input belongs to the **only big bars** partition.

Your lesson is: explore the boundaries in between your partitions!

Watch our video on Youtube:
https://www.youtube.com/embed/uP_SpXtHxoQ

Automating boundary testing with JUnit (via Parameterized Tests)

We just analysed the boundaries and derived test cases using a domain matrix. It is time to automated these tests using JUnit.

You might have noticed that in the domain matrix we always have a certain amount of input values and, implicitly, an expected output value. We could just implement the boundary tests by making a separate method for each test. However, the amount of tests can quickly become large and then this approach is not maintainable. Also, the code in these test methods will be largely the same. After all, we do the same thing with just different input and output values.

Luckily, JUnit offers a solution where we can implement the tests with just one method: Parameterized Tests. As the naming suggests, with a parameterized test we can make a test method with parameters. So we can create a test method and make it act based on the arguments we give the method. To define a parameterized test you can use the `@ParameterizedTest` annotation instead of the usual `@Test` annotation.

Now that we have a test method that has some parameters, we have to give it the values that it should execute the tests with. In general these values are provided by a `Source`. Here we focus and a certain kind of `Source`, namely the `CsvSource`. With the `CsvSource`, each test case is given as a comma separated list of input values. We give this list in a string. To execute multiple test with the same test method, the `CsvSource` expects list of strings, where each string represents the values for one test case. The `CsvSource` is an annotation itself, so in an implementation it would like like the following: `@CsvSource({"value11, value12", "value21, value22", "value31, value32", ...})`

We are going to implement the test cases that we found in the previous example using the parameterized test. Suppose we are testing a method that returns the result of the decision we analyzed in the example. To automate the tests we create a test method with three parameters: `x` , `y` , `expectedResult` . `x` and `y` are integers. The `expectedResult` is a boolean, as the result of the method is also a boolean.

```
@ParameterizedTest
@CsvSource({
  "5, 24, true",
  "4, 13, false",
  "20, -75, false",
  "19, 48, true",
  "15, 89, true",
  "8, 90, false"
})
public void exampleTest(int x, int y, boolean expectedResult) {
  Example example = new Example();

  assertEquals(expectedResult, example.run(x, y))
}
```

The assertion in the test checks if the result of the method, with the `x` and `y` values, is the expected result.

From the values you can see that each of the six test cases corresponds to one of the test cases in the domain matrix.

Watch our video on Youtube: https://www.youtube.com/embed/fFksNXJJfiE

The CORRECT way

The *Pragmatic Unit Testing in Java 8 with JUnit*, by Langr, Hunt, and Thomas, has an interesting discussion about boundary conditions. Authors call it the **CORRECT** way, as each letter represents one boundary condition you should consider:

- Conformance:
    - Many data elements must conform to a specific format. Example: e-mail (always name@domain). If you expect an e-mail, and you do not receive an e-mail, your software might crash.
    - What should we do? Test when your input is not in conformance with what is expected.
- Ordering:

  - Some inputs might come in specific orders. Imagine a system that receives different products to be inserted in a basket. The order of the data might influence the output. What happens if the list is ordered? Unordered?
  - What should we do? Make sure our program works even if the data comes in an unordered manner (or return an elegant failure to user, avoiding the crash).
- Range:
  - Inputs often should usually be within a certain range. Example: Age should always be greater than 0 and smaller than 120.
  - What should we do? Test what happens when we provide inputs that are outside of the expected range.
- Reference:
  - In OOP systems, objects refer to other objects. Sometimes these relationships get very deep. Moreover, we might depend on external dependencies. What happens if these dependencies don't behave as expected?
  - What should we do? When testing a method, consider:
    - What it references outside its scope
    - What external dependencies it has
    - Whether it depends on the object being in a certain state
    - Any other conditions that must exist
- Existence:
  - Does something really exist? What if it doesn't? Imagine you query a database, and your database returns empty. Will our software behave correctly?
  - What should we do? What happens if we something we are expecting does not really exist?
- Cardinality:
  - Basically, the famous *off-by-one* errors. In simple words, our loop performed one step less (or more) than it should.
  - What should we do? Make sure we test our loops in different situations, such as when it actually performs zero iterations, one iterations, or many. (We'll discuss more about loops in the structural-based testing chapter).
- Time
  - Many perspectives here. First, systems rely a lot on dates and times. What happens if we receive inputs that are not ordered in regards to date and time?
  - Timeouts: does our system handle timeouts well?
  - Concurrency: does our system handle concurrency well?

Watch our video on Youtube:
https://www.youtube.com/embed/oxNEUYqEvzM

Watch our video on Youtube: https://www.youtube.com/embed/PRVqsJ5fT2I

Exercises

Exercise 1. We have the following method.

```java
public String sameEnds(String string) {
  int length = string.length();
  int half = length / 2;

  String left = "";
  String right = "";

  int size = 0;
  for(int i = 0; i < half; i++) {
    left += string.charAt(i);
    right = string.charAt(length - 1 - i) + right;

    if (left.equals(right))
      size = left.length();
  }

  return string.substring(0, size);
}
```

Perform boundary analysis on the condition in the for-loop: `i < half`, i.e. what are the on- and off-point and the in- and out-points? You can give the points in terms of the variables used in the method.

Exercise 2. Perform boundary analysis on the following decision: `n % 3 == 0 && n % 5 == 0`. What are the on- and off-points?

Exercise 3. A game has the following condition: `numberOfPoints <= 570`. Perform boundary analysis on the condition. What are the on- and off-point of the condition? Also give an example for both an in-point and an out-point.

Exercise 4. We extend the game with a more complicated condition: `(numberOfPoints <= 570 && numberOfLives > 10) || energyLevel == 5`.

Perform boundary analysis on this condition. What is the resulting domain matrix?

Exercise 5. Regarding **boundary analysis of inequalities** (e.g., `a < 10`), which of the following statements **is true**?

1. There can only be a single on-point which always makes the condition true.
2. There can be multiple on-points for a given condition which may or may not make the condition true.
3. There can only be a single off-point which may or may not make the condition false.
4. There can be multiple off-points for a given condition which always make the condition false.

Exercise 6. A game has the following condition: `numberOfPoints > 1024`. Perform a boundary analysis.

Exercise 7. Which one of the following statements about the **CORRECT** principles is **true**?

1. We should suppose that external dependencies are already on the right state for the test (REFERENCE).
2. We should test different methods from the same class in an isolated way in order to avoid order issues (TIME).
3. Whenever we encounter a loop, we should always test whether the program works for 0, 1, and 10 iterations (CARDINALITY).
4. We should always test the behavior of our program when any expected data actually does not exist (EXISTENCE).

References

- Jeng, B., & Weyuker, E. J. (1994). A simplified domain-testing strategy. ACM Transactions on Software Engineering and Methodology (TOSEM), 3(3), 254-270.
- Chapter 7 of Pragmatic Unit Testing in Java 8 with Junit. Langr, Hunt, and Thomas. Pragmatic Programmers, 2015.

Structural-Based Testing

In a previous chapter, we discussed how to test software using requirements as the main artifact for guidance. In this chapter, we will use a different source of information to create tests: the source code itself. We can call the set of techniques that use the structure of the source code as a way to guide the testing, **structural-based testing** techniques.

Understanding structural-based testing techniques boils down to understand the different coverage criteria. These coverage criteria are highly related to test coverage, a concept that many developers know. By test coverage, we mean the amount (or percentage) of production code that is exercised by the tests.

We will cover the following coverage criteria:

- Line coverage (and statement coverage)
- Block coverage
- Branch/Decision coverage
- Condition (Basic and Condition+Branch) coverage
- Path coverage
- MC/DC coverage

Watch a summary of one of our lectures in structural testing!

Watch our video on Youtube: https://www.youtube.com/embed/busfqNkpgKI

Line (and statement) coverage

As the name suggests, when determining the line coverage, we look at the amount of lines of code that are covered by the tests (more specifically, by at least one test).

See the following example: We consider a piece of code that returns the points of the person that wins a game of Black jack.

```
public class BlackJack {
  public int play(int left, int right) {
1.  int ln = left;
2.  int rn = right;
3.  if (ln > 21)
4.    ln = 0;
5.  if (rn > 21)
6.    rn = 0;
7.  if (ln > rn)
8.    return ln;
9.  else
10.   return rn;
  }
}
```

The `play(int left, int right)` method receives the amount of points of two players and returns the value like specified. Now let's make two tests for this method.

```
public class BlackJackTests {
  @Test
  public void bothPlayersGoTooHigh() {
    int result = new BlackJack().play(30, 30);
    assertThat(result).isEqualTo(0);
  }

  @Test
  public void leftPlayerWins() {
    int result = new BlackJack().play(10, 9);
    assertThat(result).isEqualTo(10);
  }
}
```

Try to follow each test method and mark which lines of the production code are exercised by it. The first test executes lines 1-7, 9, and 10 as both values are higher than 21 and when the program arrives at line 7. `ln` equals `rn` so the statement `ln > rn` is `false`. This means that 9 out of the 10 lines are covered and the line coverage is $\frac{9}{10} \cdot 100\% = 90\%$ (after all, the test exercised 9 out of the 10 lines of that method). Line 8 is therefore the only line that the first test does not cover. The second test, `leftPlayerWins`, complements the first test, and executes lines 1-3, 5, 7 and 8. So when we execute both of our tests, the line coverage is 100%.

More formally, we can compute line coverage as:

$$\text{line coverage} = \frac{\text{lines covered}}{\text{lines total}} \cdot 100\%$$

Note: Defining what constitutes a line is up to the tester. One might count, for example, the method declaration as a code line. We prefer not to count the method declaration line.

Watch our video on Youtube: https://www.youtube.com/embed/rkLsvlPlOHc

Why is line coverage a bit problematic?

Using lines of code as a way to determine line coverage is a simple and straightforward idea. However, counting the covered lines is not always a good way of calculating the coverage. The amount of lines in a piece of code is heavily dependent on the programmer that writes the code. In Java, for example, you can often write a whole method in just one line (for your future colleagues' sake, please don't). In that case, the line coverage would always be 100% if you test the method.

We are again looking at Black Jack example. The `play` method can also be written in 6 lines, instead of 10:

```
public int play(int left, int right) {
1.  int ln = left;
2.  int rn = right;
3.  if (ln > 21) ln = 0;
4.  if (rn > 21) rn = 0;
5.  if (ln > rn) return ln;
6.  else return rn;
}
```

The same `leftPlayerWins` test covered $\frac{6}{10}$ lines in the first `play` method. Now, it covers lines 1-5, so $\frac{5}{6}$ lines. The line coverage went up from 60% to 83%, while testing the same method with the same test. This is definitely not ideal.

We need a better representation for source code. One that is independent of the developers' personal code styles.

Note: Some coverage tools measure coverage as statement level. Statements are the unique instructions that your JVM, for example, executes. This is a bit better, as splitting one line code in two would not make a difference, but still not good enough.

Watch our video on Youtube:
https://www.youtube.com/embed/iQECMbKLez0

Blocks and Control-Flow Graph

A Control-Flow Graph (or CFG) is an agnostic representation of a piece of code. It consists of basic blocks, decision blocks, and arrows that connect these blocks.

A basic block is composed of "all statements that are executed together", with no if or for conditions that might create different branches in the code. Basic blocks are often represented with a square. A decision block, on the other hand, represents all the statements in the source code that can create different branches. Decision blocks are often represented as diamonds.

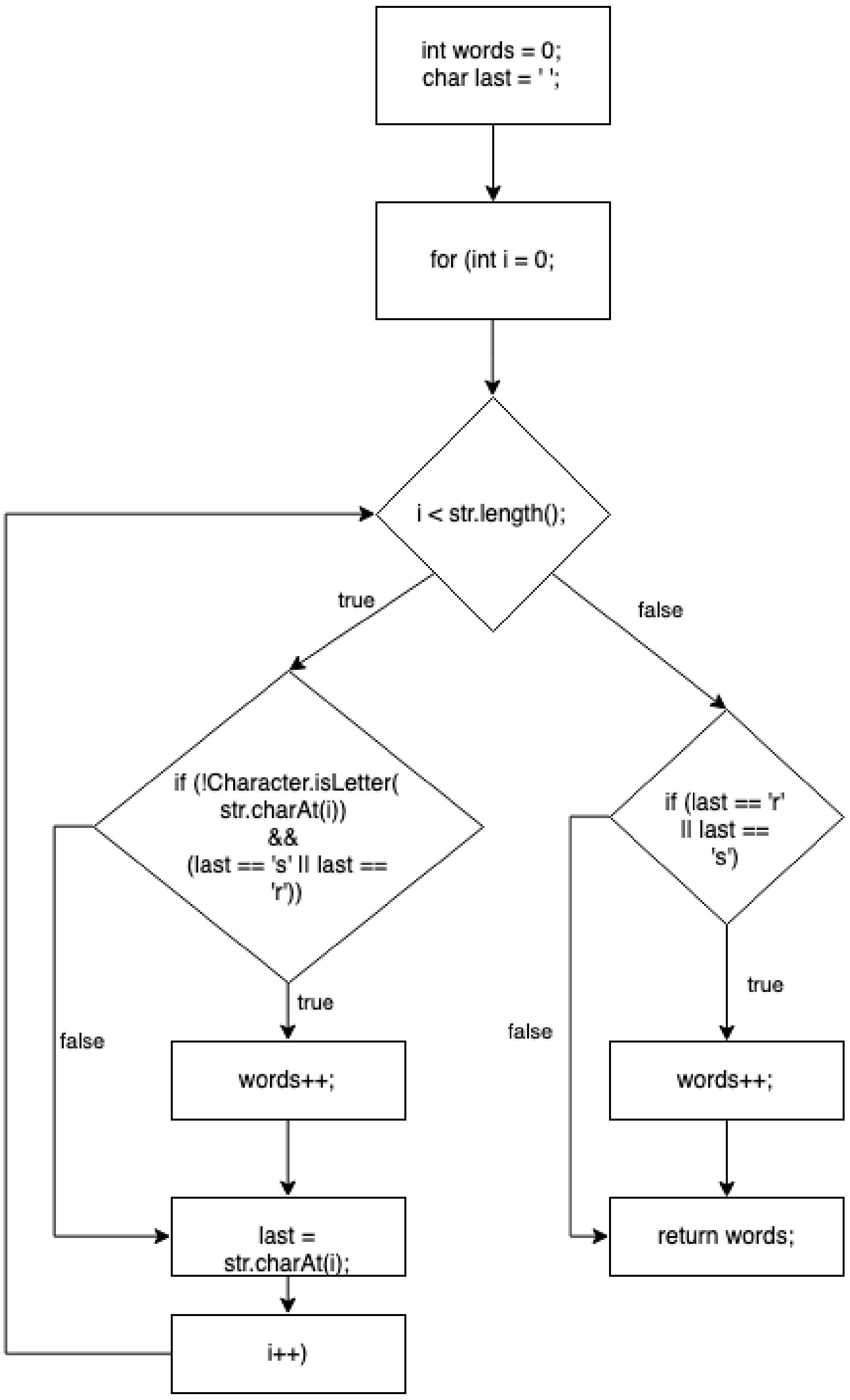
We then connect the blocks according to the flow of the program. A basic block has always a single outgoing edge; a decision block has always two outgoing edges (where you go in case of the decision being evaluated to true, and where you go in case the decision being evaluated to false).

You can see an example of a CFG below.

We write a program for the following problem: Given a sentence, you should count the number of words that end with either an "s" or an "r". A word ends when a non-letter appears.

```
public class CountLetters {
  public int count(String str) {
1.  int words = 0;
2.  char last = ' ';
3.  for (int i = 0; i < str.length(); i++) {
4.    if (!Character.isLetter(str.charAt(i))
5.        && (last == 's' || last == 'r')) {
6.      words++;
7.    }
8.    last = str.charAt(i);
9.  }
10. if (last == 'r' || last == 's')
11.   words++;
12. return words;
  }
}
```

The corresponding CFG:



Note that we split the for-loop into two blocks (variable initialization, and increment) and a decision. Every decision has one outgoing arrow for true and one for false, indicating what the program will do based on the condition. `return words;` does not have an outgoing arrow as the program stops after that statement.

Note how agnostic this CFG representation is. You can even build CFGs of program written in different languages. They might even look the same!

TODO: record a video explaining how to build CFGs

Block coverage

We can use blocks as a coverage criteria, in the same way we did with lines: instead of aiming at covering 100% of the lines, we aim at covering 100% of the blocks.

The formula to measure block coverage is similar:

$$\text{block coverage} = \frac{\text{blocks covered}}{\text{blocks total}} \cdot 100\%$$

Note that blocks do not depend on how the developer wrote the code. Thus, we will not suffer from having different coverage numbers just because the developer wrote the code in a different way.

For the `CountLetters` program, a test T1 = "cats and dogs" exercises all the blocks, and thus, reaches 100% block coverage.

Branch/Decision coverage

Complex programs often use a lot of conditions (e.g. if-statements). When testing these programs, aiming at 100% line or block coverage might not be enough to cover all the cases we want. We need a stronger criteria.

Branch coverage works the same as line and statement coverage. This time, however, we do not count lines or blocks, but the number of possible decision outcomes our program has. Whenever you have a decision block, that decision block has two outcomes. We consider our test suite to achieve 100% branch coverage (or decision coverage, as both terms mean the same) whenever we have tests exercising all the possible outcomes.

$$\text{branch coverage} = \frac{\text{decision outcomes covered}}{\text{decision outcomes total}} \cdot 100\%$$

In practice, these decisions (or branches) are easy to find in a CFG. Each arrow with true of false (so each arrow going out of a decision) is a branch.

Let's aim at 100% branch coverage for the `count` method above.

```
public class CountLettersTests {
  @Test
  public void multipleMatchingWords() {

    int words = new CountLetters()
        .count("cats|dogs");

    assertEquals(2, words);
  }

  @Test
  public void lastWordDoesntMatch() {

    int words = new CountLetters()
        .count("cats|dog");

    assertEquals(1, words);
  }
}
```

The first test (by providing `cats|dogs` as input) covers all the branches in the left part of the CFG. At the right part, it covers the top false branch, because at some point `i` will be equals to `str.length()`. Then the word `dogs` ends with an `s`, so it also covers the true branch on the right side of the CFG. This gives the test $\frac{5}{6} \cdot 100\% = 83\%$ branch coverage.

The only branch that is not covered is the false branch at the bottom right of the CFG. This branch is executed when the last word does not end with an `r` or an `s`. The second test executes this branch (by giving the word `cats|dog`) so the two tests together have a branch/decision coverage of 100%.

Watch our video on Youtube: https://www.youtube.com/embed/XiWtG8PKH-A

Note: In the video, we use *squares* to represent decision blocks. We did it just because otherwise the control flow graph would not fit in the video. When doing control flow graphs, please use *diamonds* to represent decision blocks.

(Basic) condition coverage

Branch coverage gives two branches for each decision, no matter how complicated this decision is. When a decision gets complicated, i.e., it contains more than one condition like `a > 10 && b < 20 && c < 10`, branch coverage might not be enough to test all the possible outcomes of all these decisions. Look at this example: a test T1 (a=20, b=10, c=5) and a test T2 (a=5, b=10, c=5) already covers this decision block. However, look how many other possibilities we have for this branch to be evaluated to false (e.g., T3 (a=20, b=30, c=5), ...).

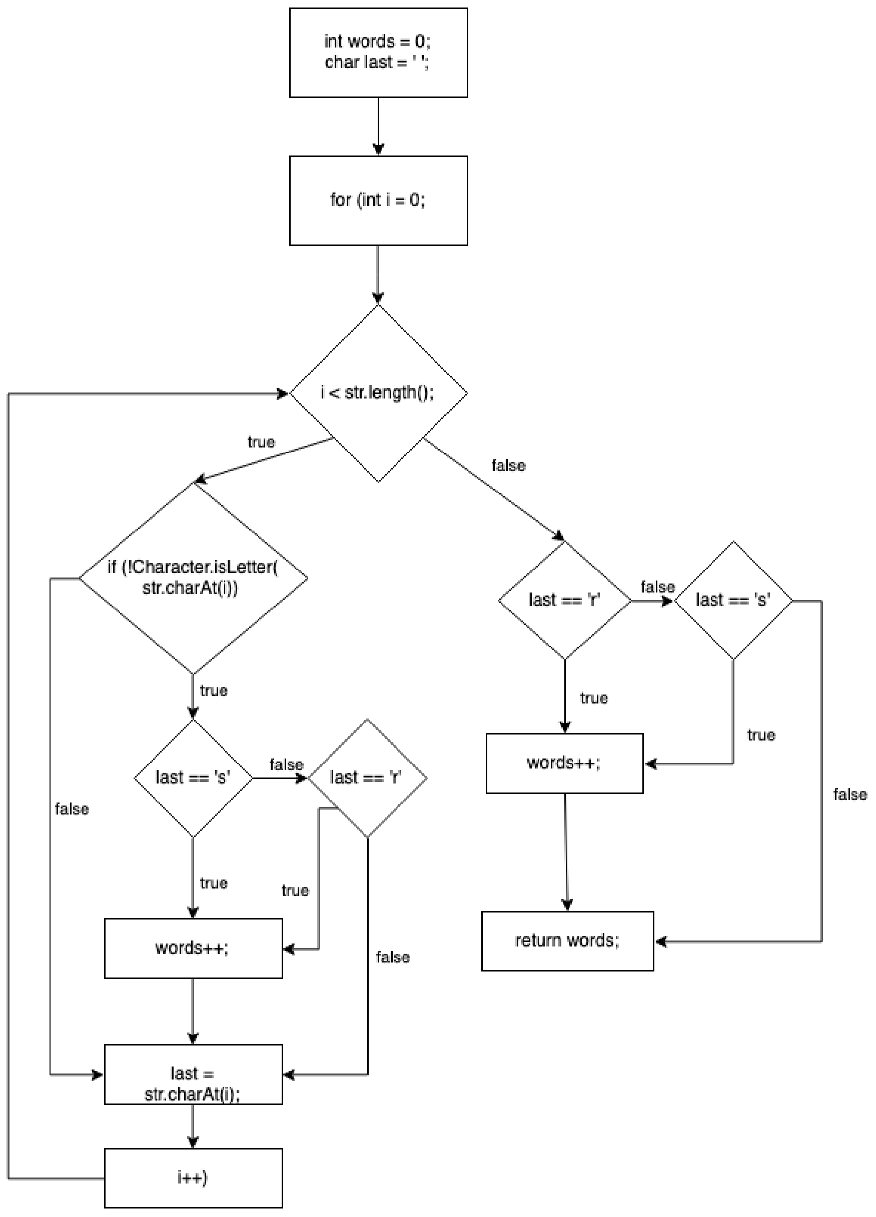
When using condition coverage as criteria, we split the decisions into single conditions. Instead of having one big decision block with the entire condition, we have multiple decision blocks, each one of one condition only. In practice, now we will exercise each condition separately, and not only the "big decision block".

As soon as you have the new CFG, it works the same as branch coverage. The formula is basically the same, but now we just have more decision outcomes to count:

$$\text{condition coverage} = \frac{\text{conditions outcome covered}}{\text{conditions outcome total}} \cdot 100\%$$

We achieve 100% condition coverage whenever all the outcomes of all the conditions in our program have been exercised. In other words, whenever all the conditions have been `true` and `false` at least once.

Once again we look at the program that counts the words ending with an "r" or an "s". Instead of branch coverage, we are interested in the condition coverage that the tests give. We start by building the more fine-grained CFG:



You can see that this new CFG has way more decision blocks than the previous one.

The first test we wrote before now covers 7 conditions and the total amount of conditions is 12. So the condition coverage is now: $\frac{7}{12} \cdot 100\% = 58\%$. This is significantly less than the 83% branch coverage, so we need more tests to get to 100% condition coverage.

Condition coverage is an improvement over the branch coverage. However, we will try to do even better in the next section.

Condition + Branch coverage

Let's think carefully about condition coverage. If we only focus on exercising the individual conditions themselves, but do not think of the overall decision, we might end up in a situation like the one below.

Imagine a `if(a > 10 && b > 20)` condition. A test `T1 = (20, 10)` makes the first condition `a > 10` to be true, and the second condition `b > 20` to be false. A test `T2 = (5, 30)` makes the first condition to be false, and the second condition to be true. Note that T1 and T2 together achieve 100% **basic condition** coverage; after all, both conditions have been exercised as true and false. However, the final outcome of the entire decision was also false! This means, we found a case where we achieved 100% basic condition coverage, but only 50% branch coverage! This is no good. This is way we called it **basic condition coverage**.

In practice, whenever we use condition coverage, we actually do **branch + condition coverage**. In other words, we make sure that we achieve 100% condition coverage (i.e., all the outcomes of all conditions are exercised) and 100% branch coverage (all the outcomes of the decisions are exercised).

From now on, whenever we mention **condition coverage**, we mean **condition + branch coverage**.

Watch our video on Youtube:
https://www.youtube.com/embed/oWPprB9GBdE

TODO: record a video showing, in a explicit way, the difference between basic condition coverage and full condition coverage

Path coverage

Finally, with condition coverage, we looked at each condition individually. This gives us way more branches to generate tests. However, note that, although we are testing each condition to be evaluated to true and to false, this does not ensure that we are testing all the paths that a program can have.

Path coverage does not consider the conditions individually; rather, it considers the (full) combination of the conditions in a decision. Each of these combinations is a path. You might see a path as a unique way to traverse the CFG. The calculation is the same as the other coverages:

$$\text{path coverage} = \frac{\text{paths covered}}{\text{paths total}} \cdot 100\%$$

See the following example. In this example we focus on a small piece of the `count` method:

```
if (!Character.isLetter(str.charAt(i))
        && (last == 's' || last == 'r')) {
    words++;
}
```

The decision of this if-statement contains three conditions and can be generalized to (A && (B || C)), with A = `!Character.isLetter(str.charAt(i))` , B = `last == 's'` and C = `last == 'r'` . To get 100% path coverage, we would have to test all the possible combinations of these three conditions. We construct a truth table to find the combinations:

Tests	A	B	C	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

This means that, for full path coverage, we would need 8 tests just to cover this if-statement. That is quite a lot for just a single statement!

By thinking about the path coverage of our test suite, we can come up of quite some good tests. The main issue is that achiving 100% path coverage might not always be feasible. The number of combinations might be too big! The amount of tests needed for full path coverage will grow exponentially with the amount of conditions in a decision.

Watch our video on Youtube: https://www.youtube.com/embed/hpE-aZYulmk

MC/DC (Modified Condition/Decision Coverage)

Modified condition/decision coverage (MC/DC from now on), looks at the combinations of conditions like path coverage does. However, instead of aiming at testing all the possible combinations, we take a certain selection process in order to identify the "important ones". Clearly, the goal of focusing on the important ones is to tackle the large amount of test cases that one needs to devise when aiming at 100% path coverage.

The idea of MC/DC is to simply exercise each condition in a way that it can, independently of the other conditions, affect the outcome of the entire decision. This might sound a bit complicated, but the example will clarify it. And interestingly, if our conditions have only a binary outcome, which is our case here, as conditions either return true or false, the number of tests we will need for that is always "only" `N+1`, where `N` is the number of conditions in the program. $N+1$ is definitely smaller than N^2!

Again, to devise a test suite that achieves 100% MC/DC coverage, we should select $N+1$ combinations of inputs where all the conditions can independently affect the outcome.

Let's do it in a mechanical way. See the example below.

Let's test the decision block we have in the previous example, with its corresponding truth table. Note how each row represents a test T_n. In this case, tests go from 1 to 8, as we have 3 decisions, and 2^3 is 8:

Tests	A	B	C	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Our goal will be to select $N+1$, in this case, $3+1=4$, tests. We go condition by condition. In this case, we start with selecting the pairs of combinations (or tests) for condition A:

- In test 1: A, B and C are all true and the outcome is true as well. We should look for another test in this table, where the value of A is flipped in comparison to test 1, but all others (B and C) are the same. In this case, we should look for a row where A=False, B=True, C=True. We find it in test 5.

Now, look at the outcome of test 5: it's false. This means we just found a pair of tests, T1 and T5, where A is the only condition that changed, and the outcome also changed. This means, we just found a pair of tests where A independently affect the outcome. Let's keep the pair {T1, T5} in our list of tests.

- Now we look at the next test. In test 2, A is again true, B is true, and C is false. We repeat the process: we search for a test where A is flipped in comparison to test 2, but B and C are the same (B=True, C=False). We find test 6. The outcome from test 6 (false) is not the same as the outcome of test 2 (true), so this means that the pair of tests {T2, T6} is also able to independently show how A can affect the final outcome.
- We repeat the process for test 3. We will find that the pair {T3, T7} is also a good one.
- We repeat the process for test 4 (A=True, B=False, C=False). Its pair is test 8 (A=False, B=False, C=False). However, note that the outcome of both tests is false. This means that the pair {T4, T8} does not really show how A can independently affect the outcome; after all, A is the only thing that changes, but the outcome is still the same...
- We now repeat it up to test 8. We will not find any other suitable pair.
- Now that we are done with condition A, we can go to condition B. And we repeat the same process, but now flipping the input of B, and keeping A and C the same.
- For T1 (A=true, B=true, C=true), we search for a test where (A=true, B=false, C=true). We find test 3. However, the outcome is the same, so the pair {T1, T3} is not a good one to show the independence of B.
- You will only find the pair {T2, T4} for combition B.
- The final condition is C. Here also only one pair of combinations will work, which is {T3, T4}. (To practice, you should do the entire process!)
- We now have all the pairs for each of the conditions:
    - A: {1, 5}, {2, 6}, {3, 7}
    - B: {2, 4}
    - C: {3, 4}
- Now we are ready to select the combinations that we want to test. For each condition (A, B, and C), we have to have at least one of the pairs. Moreover, we want to minimize the total amount of tests, and we know that we can do it with N+1 tests.
- We do not have any choices with conditions B and C, as we only found one pair for each. This means that we have to test combinations 2, 3 and 4.
- Now we need to make sure to cover a pair of A. To do so we can either add combination 6 or 7. Both are good. Let's pick, for example, 6. (Note: You can indeed have more than one set of tests that achieve 100% MC/DC; all solutions are equally valid and good!)

- The combinations that we need for 100% MC/DC coverage are {2, 3, 4, 6}. These are only 4 combinations/tests we should focus. This is a lot better than the 8 tests we needed for the path coverage.

Indeed, in the example above, we saw that we need fewer tests when using MC/DC instead of path coverage.

Watch our video on Youtube: https://www.youtube.com/embed/HzmnCVaICQ4

Loop boundary adequacy

What to do when we have loops? After all, whenever there is a loop, the block inside of the loop might be executed many times; this would make testing more complicated.

Think of a `while(true)` loop. It can go forever. If we wanted to be rigorous about it, we would have to test the program where the loop block is executed one time, two times, three times, ... Imagine a `for(i = 0; i < 10; i++)` loop with a `break` inside of the body. We would have to test what happens if the loop body executes one time, two times, three times, ..., up to ten times. It might be impossible to exhaustively test all the combinations!

What trade-off can we make? And, more especifically, for unbounded loops, where we do not really know how many times it will be executed. We can define a **loop boundary adequacy criteria**:

A test suite satisfies this criterion if and only if for every loop:

- A test case exercises the loop zero times
- A test case exercises the loop once
- A test case exercises the loop multiple times

Pragmatically speaking, the main challenge is devising tests where the loop is exercised multiple times. Devising tests that can indeed explore the space efficiently requires a good understanding of the program itself. Our tip is for you to make a little use of specification-based techniques here. If you understand the specs, you might be able to devise good tests for the particular loop.

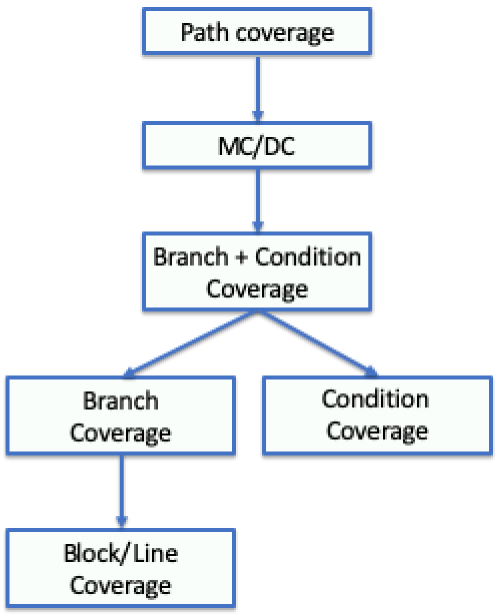
TODO: record a video about the loop boundary adequacy

Criteria subsumption

You might have noticed that, the more criteria we studied, the more "rigorous" they became. We started our discussion with line coverage. Then we discussed branch coverage, and we noticed that we could generate more tests if we focused on branchs. Then, we discussed branch + condition coverage, and we noticed that we could generate even more tests if we also focused on the conditions. And we kept doing that up to here.

There is indeed a relationship between all these criteria. Some strategies **subsume** other strategies. More formally, a strategy X subsumes strategy Y if all elements that Y exercises are also exercised by X. You can see in the figure below how the relationship among all the coverage criteria we studied.

You can see that, for example, branch coverage subsumes line coverage. This means that 100% of branch coverage always implies in 100% line coverage; however, 100% of line coverage does not imply in 100% branch coverage. 100% of branch + condition coverage imply in 100% branch coverage and 100% of line coverage.



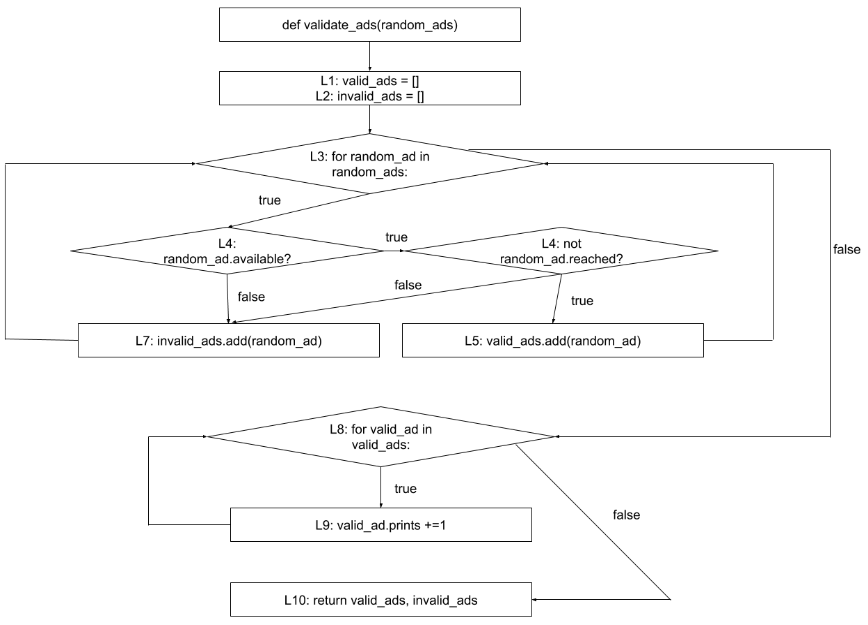
TODO: record a video about the criteria subsumption

More examples of Control-Flow Graphs

We can do Control-Flow Graphs for programs in any programming language. For example, see the piece of Python code below:

```
# random_ads is a list of ads.
# an ad contains three attributes:
# * available: true/false indicating whether the ad
#   is still available.
# * reached: true/false indicating
#   whether the number of paid prints was reached.
# * prints: an integer indicating the
#   number of times that the ad was printed.
def validate_ads(random_ads):
01. valid_ads = []
02. invalid_ads = []

03. for random_ad in random_ads:
04.   if random_ad.available and not random_ad.reached:
05.     valid_ads.add(random_ad)
06.   else:
07.     invalid_ads.add(random_ad)

08. for valid_ad in valid_ads:
09.   valid_ad.prints += 1

10. return valid_ads, invalid_ads
```

A CFG for this piece of code would look like:


def validate_ads(random_ads)
L1: valid_ads = []
L2: invalid_ads = []
L3: for random_ad in random_ads:
true
L4: random_ad.available?
true
L4: not random_ad.reached?
false
false
true
false
L7: invalid_ads.add(random_ad)
L5: valid_ads.add(random_ad)
L8: for valid_ad in valid_ads:
true
false
L9: valid_ad.prints +=1
L10: return valid_ads, invalid_ads

Study tip: Note how we modelled the `for each` loop.

How to use structural testing in practice

As a tester, you use the different coverage criteria to derive tests. If you decide that your goal is to achieve at least 80% branch + condition coverage, you derive tests until you reach it.

Is there any advantage in using structural testing? We refer to two papers:

- Hutchins et al.: "Within the limited domain of our experiments, test sets achieving coverage levels over 90% usually showed significantly better fault

detection than randomly chosen test sets of the same size. In addition, significant improvements in the effectiveness of coverage-based tests usually occurred as coverage increased from 90% to 100%. However, the results also indicate that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set."
- Namin and Andrews: "Our experiments indicate that coverage is sometimes correlated with effectiveness when size is controlled for, and that using both size and coverage yields a more accurate prediction of effectiveness than size alone. This in turn suggests that both size and coverage are important to test suite effectiveness."

For interested readers, a extensive literature review on the topic can be found in Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys (csur), 29(4), 366-427.

Exercises

For the first couple of exercises we use the following code:

```
public boolean remove(Object o) {
01.  if (o == null) {
02.    for (Node<E> x = first; x != null; x = x.next) {
03.      if (x.item == null) {
04.        unlink(x);
05.        return true;
         }
       }
06.  } else {
07.    for (Node<E> x = first; x != null; x = x.next) {
08.      if (o.equals(x.item)) {
09.        unlink(x);
10.        return true;
         }
       }
     }
11.  return false;
}
```

This is the implementation of JDK8's LinkedList remove method. Source: OpenJDK.

Exercise 1. Give a test suite (i.e. a set of tests) that achieves 100% **line** coverage on the `remove` method. Use as few tests as possible.

The documentation on Java 8's LinkedList methods, that may be needed in the tests, can be found in its Javadoc.

Exercise 2. Create the Control Flow Graph (CFG) for the `remove` method.

Exercise 3. Look at the CFG you just created. Which of the following sentences **is false**?

1. A minimal test suite that achieves 100% basic condition coverage has more test cases than a minimal test suite that achieves 100% branch coverage.

2. The method `unlink()` is for now treated as an 'atomic' operation, but also deserves specific test cases, as its implementation might also contain decision blocks.
3. A minimal test suite that achieves 100% branch coverage has the same number of test cases as a minimal test suite that achieves 100% full condition coverage.
4. There exists a single test case that, alone, is able to achieve more than 50% of line coverage.

Exercise 4. Give a test suite (i.e. a set of tests) that achieves 100% **branch** coverage on the `remove` method. Use as few tests as possible.

The documentation on Java 8's LinkedList methods, that may be needed in the tests, can be found in its Javadoc.

Exercise 5. Consider the decision `(A or C) and B` with the corresponding decision table:

Decision	A	B	C	(A | C) & B
1	T	T	T	T
2	T	T	F	T
3	T	F	T	F
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

What is the set with the minimum amount of tests needed for 100% MC/DC (Modified Condition / Decision Coverage)?

For the next three exercises use the code below. This method returns the longest substring that appears at both the beginning and end of the string without overlapping. For example, `sameEnds("abXab")` returns `"ab"`.

```java
public String sameEnds(String string) {
  int length = string.length();
  int half = length / 2;

  String left = "";
  String right = "";

  int size = 0;
  for (int i = 0; i < half; i++) {
    left = left + string.charAt(i);
    right = string.charAt(length - 1 - i) + right;

    if (left.equals(right)) {
      size = left.length();
    }
  }

  return string.substring(0, size);
}
```

This code is based on the same ends problem.

Exercise 6. Draw the Control Flow Graph of the source code above.

Exercise 7. Give a test case (by the input string and expected output) that achieves 100% line coverage.

Exercise 8. Given the source code of the `sameEnds` method. Which of the following statements is **not correct**?

1. It is possible to devise a single test case that achieves 100% line coverage and 100% decision coverage.
2. It is possible to devise a single test case that achieves 100% line coverage and 100% (basic) condition coverage.
3. It is possible to devise a single test case that achieves 100% line coverage and 100% decision + condition coverage.
4. It is possible to devise a single test case that achieves 100% line coverage and 100% path coverage.

Now consider this piece of code for the FizzBuzz problem. Given an `int n`, it returns the string form of the number followed by "!". So the int 6 would yield "6!". Except if the number is divisable by 3 it returns "Fizz!" and if it is divisable by 5 it returns "Buzz!". If the number is divisable by both 3 and 5 it returns "FuzzBuzz!" Based on a CodingBat problem

```java
public String fizzString(int n) {
  if (n % 3 == 0 && n % 5 == 0)
       return "FizzBuzz!";
  if (n % 3 == 0)
      return "Fizz!";
  if (n % 5 == 0)
      return "Buzz!";
  return n + "!";
}
```

Exercise 9. Assume we have two test cases with an input integer: T1 = 15 and T2 = 8.

What is the condition coverage these test cases give combined?

What is the decision coverage?

The next couple of exercises use Java's implementation of the LinkedList's `computeIfPresent()` method.

```java
public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
01. if (rf == null) {
02.   throw new NullPointerException();
    }

03. Node<K,V> e;
04. V oldValue;
05. int hash = hash(key);
06. e = getNode(hash, key);
07. oldValue = e.value;

08. if (e != null && oldValue != null) {

09.   V v = rf.apply(key, oldValue);

10.   if (v != null) {
11.     e.value = v;
12.     afterNodeAccess(e);
13.     return v;
      }
      else {
14.     removeNode(hash, key, null, false, true);
      }
    }
15. return null;
}
```

Exercise 10. Draw the Control Flow Graph (CFG) of the method above.

Exercise 11. How many tests do we need **at least** to achieve 100% line coverage?

Exercise 12. How many tests do we need **at least** to achieve 100% branch coverage?

Exercise 13. Which of the following statements concerning the subsumption relations between test adequacy criteria **is true**:

1. MC/DC subsumes statement coverage.
2. Statement coverage subsumes branch coverage.
3. Branch coverage subsumes path coverage.
4. Basic condition coverage subsumes branch coverage.

Exercise 14. A test suite satisfies the loop boundary adequacy criterion if for every loop L:

1. Test cases iterate L zero times, once, and more than once.
2. Test cases iterate L once and more than once.

3. Test cases iterate L zero times and one time.
4. Test cases iterate L zero times, once, more than once, and N, where N is the maximum number of iterations.

Exercise 15. Consider the expression `((A and B) or C)`. If we aim to achieve 100\% \emph{Modified Condition / Decision Coverage} (MC/DC), the **minimum** set of tests we should select is:

1. {2, 3, 4, 6}
2. {1, 3, 4, 6}
3. {2, 3, 5, 6}
4. {3, 4, 7, 8}

References

- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Chapter 12 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys (csur), 29(4), 366-427.
- Cem Kaner on Code Coverage: http://www.badsoftware.com/coverage.htm
- Arie van Deursen on Code Coverage: http://avandeursen.com/2013/11/19/test-coverage-not-for-managers/
- Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994, May). Experiments of the effectiveness of data flow-and control flow-based test adequacy criteria. In Proceedings of the 16th international conference on Software engineering (pp. 191-200). IEEE Computer Society Press.
- Namin, A. S., & Andrews, J. H. (2009, July). The influence of size and coverage on test suite effectiveness. In Proceedings of the eighteenth international symposium on Software testing and analysis (pp. 57-68). ACM.

Model-Based Testing

In model based testing, we use models of the system to derive tests. In this chapter we briefly show what a model is (or can be), and go over some of the models used in software testing. The two models covered in this chapter are decision tables and state machines.

Models

In software testing, a model is a simpler way of describing the program under test. A model holds some of the attributes of the program that the model was made of. Given that the model preserves some of the original attributes of the system under test, it can be used to analyse and test the system.

Why should we use models at all? A model gives us a structured way to understand how the program operates (or should operate).

Watch our video on Youtube: https://www.youtube.com/embed/5yuFf4-4JnE

Decision Tables

Decision tables are used to model how a combination of conditions should lead to a certain action. These tables are easy to understand and can be validated by the client that the software is created for. Developers can use the decision tables to derive tests that verify the correct implementation of the requirements with respect to the conditions.

Creating decision tables

A decision table is a table containing the conditions and the actions performed by the system based on these conditions. In this section, we'll discuss how to build them in first place.

For now the table contains all the combinations of conditions explicitly. Later we will look at ways to bring the amount of combinations present in the table down a bit.

In general a decision table looks like the following:

		Variants			
Conditions	<Condition1>	T	T	F	F
	<Condition2>	T	F	T	F
Action	<Action>	value1	value2	value3	value4

The chosen conditions should always be independent from one another. In this type of decision tables, the order of the conditions also do not matter, e.g., making `<Condition2>` true and `<Condition1>` false or making `<Condition1>` false and after that `<Condition2>` true, should result on the same outcome. (If the order does matter in some way, a state machine might be a better model. We cover state machines later in this chapter.)

When choosing a phone subscription, there are a couple of options you could choose. Depending on these options a price per month is given. We consider the two options:

- International services
- Auto-renewal

In the decision table these will be turned into conditions. True then corresponds to a chosen option and false corresponds to a option that is not chosen. Taking international should increase the price per month. Auto-renewal decreases the price per month.

The decision tables is the following:

		Variants			
Conditions	International	F	F	T	T
	Auto-renewal	T	F	T	F
Action	price/month	10	15	30	32

You can see the different prices for the combinations of international and auto-renewal.

Don't Care values: In some cases the value of a condition might not influence the action. This is represented as a don't care value, often abbreviated to "dc".

Essentially, "dc" is an combination of two columns. These two columns have the same values for the other conditions and the same result. Only the condition that had the dc value has different values in the expanded form.

		Variants		
Conditions	<Condition1>	T	dc	F
	<Condition2>	dc	T	F
Action	<Action>	value1	value1	value2

can be expanded to:

		Variants			
Conditions	<Condition1>	T	T	T	F
	<Condition2>	T	F	T	T
Action	<Action>	value1	value1	value1	value1

After expanding we can remove the duplicate columns. We end up with the decision table below:

		Variants			
Conditions	<Condition1>	T	T	F	F
	<Condition2>	T	F	T	F
Action	<Action>	value1	value1	value1	value2

We add another condition to the example above. A loyal costumer receives the same discount as a costumer who chooses the auto-renewal option. However, a costumer only gets the discount from one of the two. The new decision table is below:

		Variants					
Conditions	International	F	F	F	T	T	T
	Auto-renewal	T	dc	F	T	dc	F
	Loyal	dc	T	F	dc	T	F
Action	price/month	10	10	15	30	30	32

Note that when auto-renewal is true, the loyal condition does not change the outcome anymore and vice versa. *Default behavior*: Usually, N conditions lead to 2^N combinations or columns. Often, however, the number of columns that are specified in the decision table can be smaller. Even if we expand all the dc values. This is done by using a default action. A default action means that if a combination of condition outcomes is not present in the decision table, the default action should be the result. If we set the default charge rate to 10 per month the new decision table can be a bit smaller:

		Variants			
Conditions	International	F	T	T	T
	Auto-renewal	F	T	dc	F
	Loyal	F	dc	T	F
Action	price/month	15	30	30	32

Watch our video on Youtube: https://www.youtube.com/embed/1u1qfJ2IrpU

Testing decision tables

Using the decision tables we can derive tests, such that we test whether the expected logic is implemented correctly. There are multiple ways to derive tests for a decision table:

- **All explicit variants:** Derive one test case for each column. The amount of tests is the amount of columns in the decision table.
- **All possible variants:** Derive a test case for each possible combination of condition values. For N conditions this leads to 2^N test cases. Often, this

approach is unrealistic because of the exponential relation between the number of conditions and the number of test cases.
- **Every unique outcome / All decisions:** One test case for each unique outcome or action. The amount of tests depends on the actions in the decision table.
- **Each condition T/F:** Make sure that each conditions is true and false at least once in the test suite. This often results in two tests: All conditions true and all conditions false.

MC/DC

One more way to derive test cases from a decision table is by using Modified Condition / Decision Coverage (MC/DC). This is a combination of the last two ways of deriving tests shown above.

We have already discussed MC/DC in the Structural-Based Testing chapter. MC/DC has the two characteristics of All devisions and Each condition T/F with an additional characteristic that makes MC/DC special:

1. Each condition is at least once true and once false in the test suite
2. Each unique action should be tested at least once
3. Each condition should individually determine the action or outcome

The third point is realized by making two test cases for each condition. In these two test cases, the condition under test should have a different value, the outcome should be different, and the other conditions should have the same value in both test cases. This way the condition that is under test individually influences the outcome, as the other conditions stay the same and therefore do not influence the outcome.

By choosing the test cases efficiently MC/DC needs less tests than all variants, while still exercising the important parts of the system. Less tests of course means less time taken to write the tests and a faster execution of the test suite.

To derive the tests we expand and rearrange the decision table of the previous example:

		v1	**v2**	**v3**	**v4**	**v5**	**v6**
	International	T	T	T	T	F	F
Conditions	Auto-renewal	T	T	F	F	T	T
	Loyal	T	F	T	F	T	F
Action	price/month	30	30	30	32	10	10

First, we look at the first condition and we try to find pairs of combinations that would cover this condition according to MC/DC. We look for combinations where only International and the price/month changes.

The possible pairs are: {v1, v5}, {v2, v6}, {v3, v7} and {v4, v8}. Testing both combinations of any of these pairs would give MC/DC for the first condition.

Moving to Auto-renewal we find the pairs: {v2, v4}, {v6, v8}. For this condition {v1, v3} and {v5, v7} are not viable pairs, because the action is the same among the two combinations.

The last condition, Loyal, gives the following pairs: {v3, v4}, {v7, v8}.

By choosing the test cases efficiently we should be able to achieve full MC/DC by choosing four of the combinations. We want to cover all the actions in the test suite. Therefore we need at least v4 and v8. With these decisions we have covered the International condition as well. Now we need one of v1, v2, v3 and one of v5, v6, v7. To cover Loyal we add v7 and to cover Auto-renewal we add v2. Now we also cover all the possible actions.

Now, for full MC/DC, we test the decisions: v2, v4, v7, v8.

Watch our video on Youtube:
https://www.youtube.com/embed/TxAFPJx6yKI

Implementing automated test cases for decision tables

Now that we know how to derive the test cases from the decision tables, it is time to implement them as automated test cases.

The most obvious way to test the combinations is to create a single test for each of the conditions.

We continue with the example we created the decision table for earlier. To start we write the tests for combinations v2 and v3. Assuming that we can use some implemented methods in a PhonePlan class the tests look like this:

```
@Test
public void internationalAutoRenewalTest() {
  PhonePlan plan = new PhonePlan();

  plan.setInternational(true);
  plan.setAutoRenewal(true);
  plan.setLoyal(false);

  assertEquals(30, plan.pricePerMonth());
}

@Test
public void internationalLoyalTest() {
  PhonePlan plan = new PhonePlan();

  plan.setInternational(true);
  plan.setAutoRenewal(false);
  plan.setLoyal(true);

  assertEquals(30, plan.pricePerMonth());
}
```

As you can see in the example above, the different tests for the different combinations are very similar. The tests do the exact same thing, but just with different values. To avoid the code duplication that comes with this approach to implementing decision table tests, we can use parameterized tests.

Parameterized Testing

Instead of implementing a single test for each of the combinations, we want to implement a single test for all the combinations. Then for each combination we execute this test with the correct values. This is done with a parameterized test.

In JUnit 5 this is done using the `ParameterizedTest` annotation. Additionally, a `Source` is needed, which is given by another annotation. In general this will look like the following:

```
@ParameterizedTest
@CsvSource({
  "true, 5, 8.0, ...",  // test 1
  "false, 2, 0.6, ...", // test 2
  "true, 1, 5.3, ...",  // test 3
  "true, 3, 4.7, ..."   // test 4
})
public void someTest(boolean param1, int param2, double param3, ...) {
  // Arrange
  SomeObject some = new SomeObject();

  // Act
  double result = some.method(param1, param2);

  // Assert
  assertEquals(param3, result);
}
```

For testing the combinations out of decision tables, usually the `CsvSource` is most convienient. Other sources can be found in the JUnit 5 docs. Each string in the `CsvSource` gives one test. Such a string consists of the arguments for the test function separated by commas. The first value is the first argument, the second value the second argument etc.

With the parameterized test we can easily create all tests we need for MC/DC of the decision table in the previous examples.

```
@ParameterizedTest
@CsvSource({
  "true, true, false, 30",   // v2
  "true, false, true, 30",   // v3
  "true, false, false, 32",  // v4
  "false, false, false, 15"  // v8
})
public void pricePerMonthTest(boolean international, boolean autoRenewal,
    boolean loyal, int price) {

  PhonePlan plan = new PhonePlan();

  plan.setInternational(international);
  plan.setAutoRenewal(autoRenewal);
  plan.setLoyal(loyal);

  assertEquals(price, plan.pricePerMonth());
}
```

You can see that the test is very similar as the tests in the previous example. Instead of directly using the values for one combination we use the parameters with the `CsvSource` to execute multiple tests.

Watch our video on Youtube: https://www.youtube.com/embed/tzcjDhdQfvM

Non-binary choices and final guidelines

TODO: Write the video's accompanying text

Watch our video on Youtube: https://www.youtube.com/embed/RHB_HaGfNjM

State Machines

The state machine is a model that describes the software system by describing its states. A system often has multiple states and various transitions between these states. The state machine model uses these states and transitions to illustrate the system's behavior.

The main focus of a state machine is, as the name suggests, the states of a system. So it is useful to think about what a state actually is. The states in a state machine model describe where a program is in its execution. If we need X to happen before we can do Y, we can use a state. X would then cause the transition to this state. From the state we can do Y, as we know that in this state X has already happened. We can use as many states as we need to describe the system's behavior well.

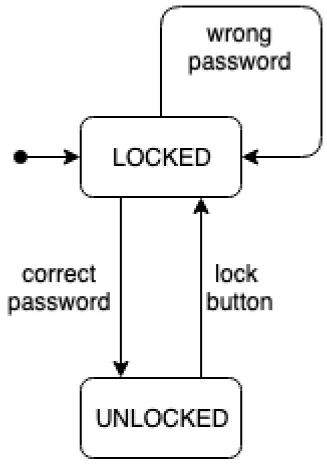
Besides states and transitions, a state machine has an initial state and events. The initial state is the state that the system starts in. From that state the system can transition to other states. Each transition is paired with an event. This event is usually one or two words that describe what has to happen to make the transition.

Of course there are some agreements on how to make the models. The notation we use is the Unified Modeling Language, UML. For the state diagrams that means we use the following symbols:

- State: $\boxed{\text{<State>}}$
- Transition: $\longrightarrow$
- Event: $\xrightarrow{\text{<Event>}}$
- Initial state: $\bullet\!\!\longrightarrow$

For the coming examples we model a (part of a) phone. We start very simple with a state machines that models the phone's ability to be locked or unlocked.

A phone that can be either locked or unlocked has two states: locked and unlocked. Before all the face recognition and fingerprint sensors, you had to enter a password to unlock the phone. A correct password unlocks the phone and if an incorrect password is given the phone stays locked. Finally, an unlocked phone can be locked again by pushing the lock button. We can use these events in the state machine.

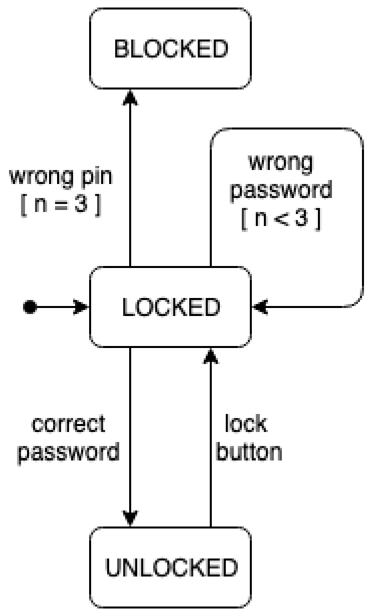


In the diagram the initial state is `LOCKED`. Usually when someone starts using their phone, it is locked. Therefore the initial state of the state machine should also be `LOCKED`.

Sometimes an event can lead to multiple states, depending on a certain condition. To model this in the state machines, we use conditional transitions. These transitions are only performed if the event happens and if the condition is true. The conditions often depend on a certain value used in the state machine. To modify these values when a transition is taken in the state machine we use actions. Actions are associated with a transition and are performed when the system uses that transition to go into another state. The notation for conditions and actions is as follows:

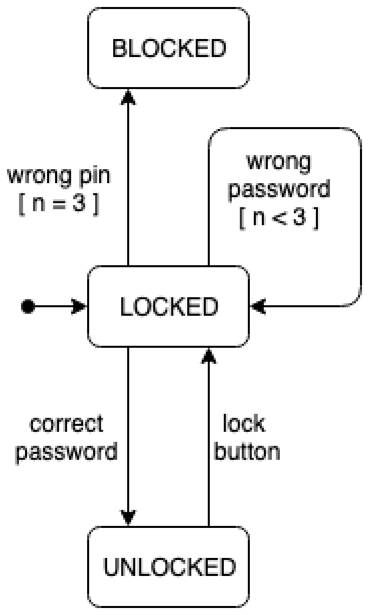
- Conditional transition: $\xrightarrow{\substack{\text{<Event>} \\ \text{[<Condition>]}}}$
- Action: $\xrightarrow{\substack{\text{<Event>} \\ \text{/ <Action>}}}$

When a user types the wrong password for four times in a row, the phone gets blocked. We use `n` in the model to represent the amount of failed attempts. Let's look at the conditional transitions that we need to model this behavior first.



When `n` (the number of failed unlock attempts) is smaller than 3, the phone stays in `LOCKED` state. However, when `n` is equal to 3, the phone goes to `BLOCKED`. Here we have an event, wrong password, than can lead to different states based on the condition.

In the previous state machine, `n` never changes. This means that the phone will never go in its `BLOCKED` state, as that requires `n` to be equal to 3. We can add actions to the state machine to make `n` change correctly.



The added actions are setting `n` to `n+1` when an incorrect password is given and to 0 when a correct password is given. This way the state machine will be in the `BLOCKED` state when a wrong password is given for four times in a row.

Watch our video on Youtube: https://www.youtube.com/embed/h4u9k-P3W0U

Watch our video on Youtube: https://www.youtube.com/embed/O1_oC-7I5E4

Testing state-machines

Like with the decision tables, we want to use the state machine model to derive tests for our software system. First, we will have a look at what might be implemented incorrectly.

An obvious error that can be made is a transition going to the wrong state. This will cause the system to act incorrectly so we want the tests to catch such errors. Additionally, the conditions in conditional transitions and the actions in transition can be wrong. Finally, the behavior of a state should stay the same at all times. This means that moving from and to a state should not change the behavior of that state.

For state machines we have a couple of test coverages. In this chapter we go over three mainly used ways of defining test coverage:

- **State coverage:** each state has to be reached at least once
- **Transition coverage:** each transition has to be exercised at least once

- **Paths:** not exactly a way of describing test coverage, but we use paths to derive test cases

To achieve the state coverage we generally bring the system in a state through transitions and then assert that the system is in that state. To test a single transition (for transition coverage) a bit more steps are needed:

1. Bring system in state that the transition goes out of
2. Assert that the system is indeed in that state
3. Trigger the transition's event.
4. If there is an action: check if this action has happened
5. Assert that the system now is in the new state that the transition points to

To achieve full state coverage we need to arrive in each state once. For the phone example we have three states so we can make three tests.

- Check that the system is `LOCKED` when it is started
- Give the correct password and check that the system is `UNLOCKED`
- Give an incorrect password four times and check that the system is `BLOCKED`

With these three tests we achieve full state coverage, as the system is in each state at some point.

With the tests above, we have covered most of the transitions as well. The only untested transition is the `lock button` from `UNLOCKED` to `LOCKED`. To test this transition, we bring the system in `UNLOCKED` by giving the correct password. Then we trigger the `lock button` and assert that the system is in `LOCKED`.

Paths and Transition trees

Besides the individual transitions, we can also test the combinations of transitions. These combinations of transitions are called paths.

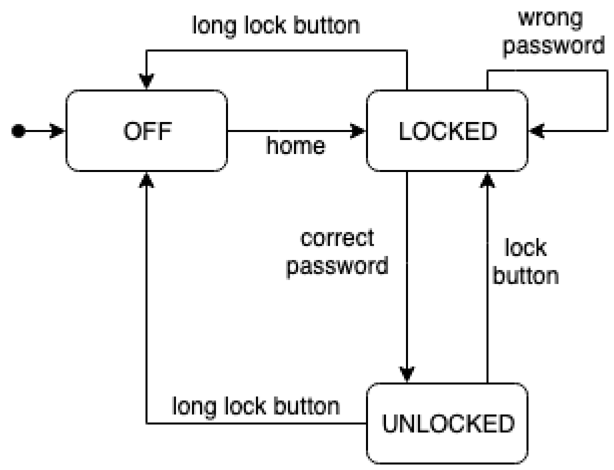
A logical thought might be: Let's test all the paths in the state machine! While this looks like a good objective, the number of paths will most likely be too high. Take a state machine that has a loop, i.e., a transition from state X to Y and a transition from state Y to X. When creating paths we can keep going back and forth these two states. This leads to an infinite amount of paths. Obviously, we cannot test all the paths. We will need to take a different approach.

The idea is that when using paths to derive test cases, we want each loop to be executed once. This way we have a finite amount of paths to create test cases for. We derive these tests by using a transition tree, which spans the graph of the state machine. Such a transition tree is created as follows:

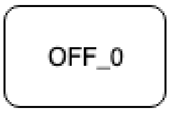
1. The root node is named as the initial state of the state machine
2. For each of the nodes at the lowest level of the transition tree:
    - If the state that the node corresponds to has not been covered before: For each of the outgoing transitions of this node's state: Add a child node that has the name of the state the transition points to. If this state is already in the tree, add or increment a number after the state's name to keep the node unique
    - If any nodes were added: Repeat from step 2.

This is also demonstrated in the example below.

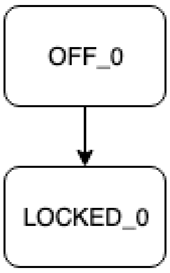
To make the transition table a bit more interesting we modify the phone's state machine to have an `OFF` state instead of a `BLOCKED` state. See the state machine below:



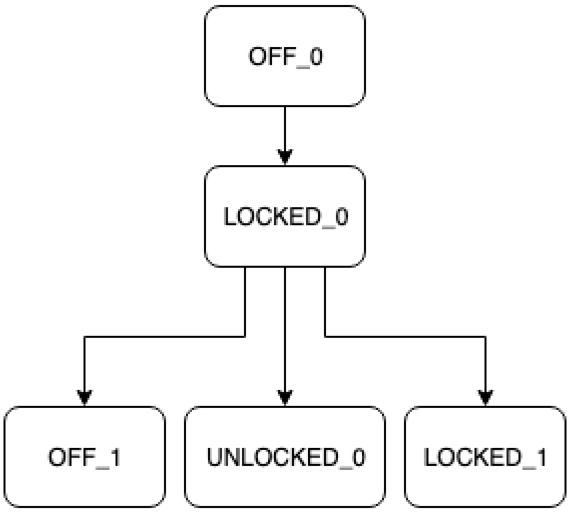
The root node of the transition tree is the initial state of the state machine. We append a number to make it easier to distinguish this node from other nodes of the same state.



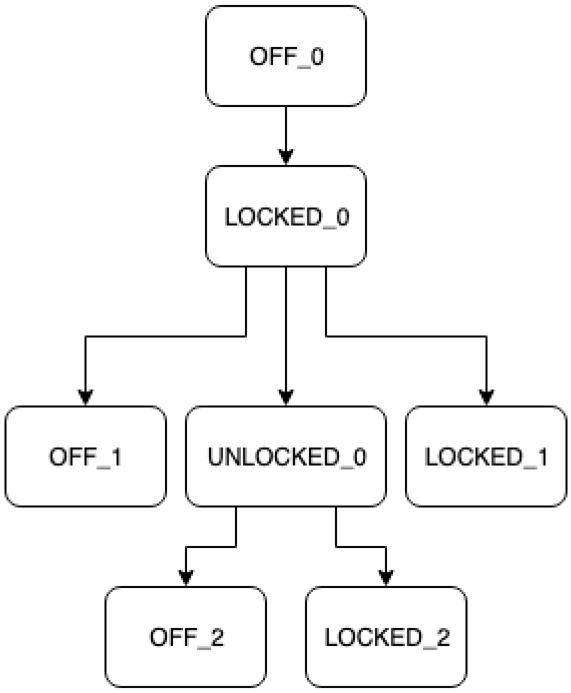
Now we for each outgoing transition from the `OFF` state we add a child node to `OFF_0` .



One node was added, so we continue by adding children to that node.



Now, the only state we have not seen yet is `UNLOCKED` in the `UNLOCKED_0` node. Therefore this is the only node we should add children to.



Now all the states of the nodes in the lowest layer have been visited before so the transition tree is done.

From a transition tree, we can derive tests. Each leaf node in the transition tree represents one path to test. This path is given by going from the root node to this leaf node. In the tests, we typically assert that we start in the correct state. Then, we trigger the next event that is needed for the given path and assert that we are in the next correct state. These events that we need to trigger can be found in the state machine. Then this is done until the whole path is followed.

In the transition tree of the previous example there are four leaf nodes: `OFF_1` , `OFF_2` , `LOCKED_1` , `LOCKED_2` . We want a test for each of these leaf nodes, that follows the path leading to that node. For `OFF_1` the test should 'move' the system from `OFF` to `LOCKED` and again to `OFF` . Looking at the state machine this gives the events `home` , `long lock button` . In the test we would assert that the system is in `OFF` , then trigger `home` , assert that the system is in `LOCKED` , trigger `long lock button` and finally assert that the system is in `OFF` .

The tests for the other three paths can be derived in similar fashion.

Using the transition tree, each loop that is in the state machine is executed once. Now the amount of tests are manageable, while testing most of the important paths in the state machine.

Watch our video on Youtube:
https://www.youtube.com/embed/pvFPzvp5Dk0

Sneak paths and Transition tables

In the previous section, we discussed transition trees and how to use them to derive tests. These tests check if the system behaves correctly when following different paths in the state machine. With this way of testing, we check if the existing transitions in a state machine behave correctly. We do not check if there exists any more transitions, transitions that should not be there. We call these paths, "sneak paths".

A **sneak path** is a path in the state machine that should not exist. So, for example, we have state X and Y and the system should not be able to transition directly from X to Y. If the system can in some way transition directly from X to Y, we have a sneak path. Of course, we want to test if such sneak paths exist in the system. To this end we make use of transition tables.

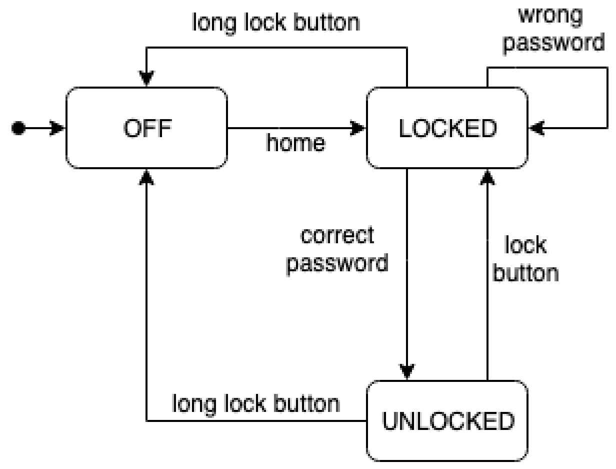
A transition table is a table containing each transition that is in the state machine. The transition is given by the state it's going out of, the event that triggers the transition, and the state the transition goes to. A transition table typically looks somewhat like the following:

STATES	Events		
	event1	event2	event3
STATE1	STATE1		STATE2
STATE2		STATE1	

We construct a transition table as follows:

- List all the state machine's states along the rows
- List all events along the columns
- For each transition in the state machine note its destination state in the correct cell of the transition table.

We take a look at the same state machine we created a transition table for:



To make the transition table we list all the states and events in the table:

STATES	Events				
	home	wrong password	correct password	lock button	long lock button
OFF					
LOCKED					
UNLOCKED					

Then we fill the table with the states that the transitions in the state machine point to:

STATES	Events				
	home	wrong password	correct password	lock button	
OFF	LOCKED				
LOCKED		LOCKED	UNLOCKED		
UNLOCKED				LOCKED	

We can see that there is, for example, a transition from `UNLOCKED` to `LOCKED` when the event `lock button` is triggered.

Now that we have the transition table, we have to decide the intended behavior for the cells that are empty. The default is to just ignore the event and stay in the same state. In some cases one might want the system to throw an exception. These decisions depend on the project and the customer's needs.

As discussed earlier, we can use the transition table to derive tests for sneak paths. Usually, we want the system to remain in its current state when we trigger an event that has an empty cell in the transition table. To test for all possible sneak paths, we create a test case for each empty cell in the transition table. This test will first bring the system to the state corresponding to the empty cell's row (you can use the transition table to find a suitable path), then triggers the event that corresponds to the empty cell's column, and finally the test asserts that the system is in the same state as before triggering the event. The amount of 'sneak path tests' is the amount of empty cells in the transition table.

With these tests we can verify both existing and non-existing paths. These techniques combined give a good testing suite from a state machine. So far, we looked at rather simple and small state machines.

Watch our video on Youtube:
https://www.youtube.com/embed/EMZB2IZT8WA

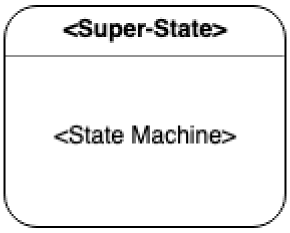
Super states and regions

When the modeled system becomes large and complex, typically so does the state machine. At some point the state machine will consist of a lot of states and transitions, which makes it unclear and impractical to work with. To resolve this issue and make a state machine more scalable we can use super states and regions.

Super states

A super state is a state that consists of a state machine. Basically, we wrap a state machine in a super-state which we can then use as a state in another state machine.

The notation of the super-state is as follows:

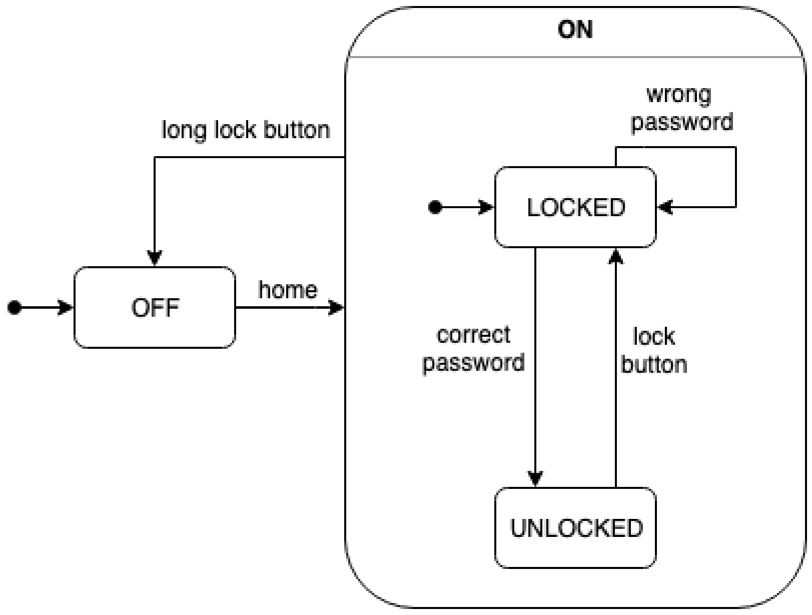


Because the super state is, in essence, a state machine that can be used as a state, we know what should be inside of a super state. The super state generally consists of multiple states and transitions, and it always has to have an initial state. Any transition going into the super state essentially goes to the initial state of the super state. A transition going out of the super state means that if the event on this transition is triggered in any of the super state's states, the system transitions into the state this transition points to.

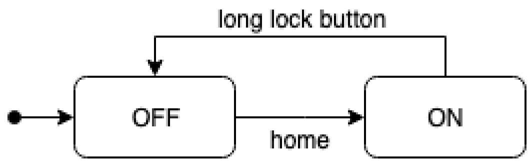
With the super state we can choose to show it fully or we can collapse it. A collapsed super state is just a normal state in the state machine. This state has the super state's name and the same incoming and outgoing transitions as the super state.

With the super states and the collapsing of super states we can modularize and combine state machines. This allows us to shift the state machine's focus to different parts of the system's behavior.

We can use a super state even in the small example of a phone's state machine. The two states `LOCKED` and `UNLOCKED` both represent the system in some sort of `ON` state. We can use this to create a super state called `ON` .



Now we can also simplify the state machine by collapsing the super state:

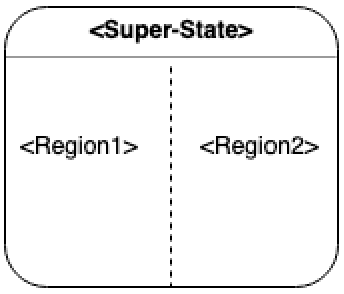


Regions

So far we have had super states that contain one state machine. Here, the system is in only one state of the super state at once. In some cases it may be useful to allow the system to be in multiple states at once. This is done with regions.

A super state can be split up into multiple regions. These are orthogonal regions, meaning that the state machines in the regions are independent from each other; they do not influence the state machines in other regions. Each region contains one state machine. When the systems enters the super state, it enters all the initial states of the regions. This means that the system is in multiple states at once.

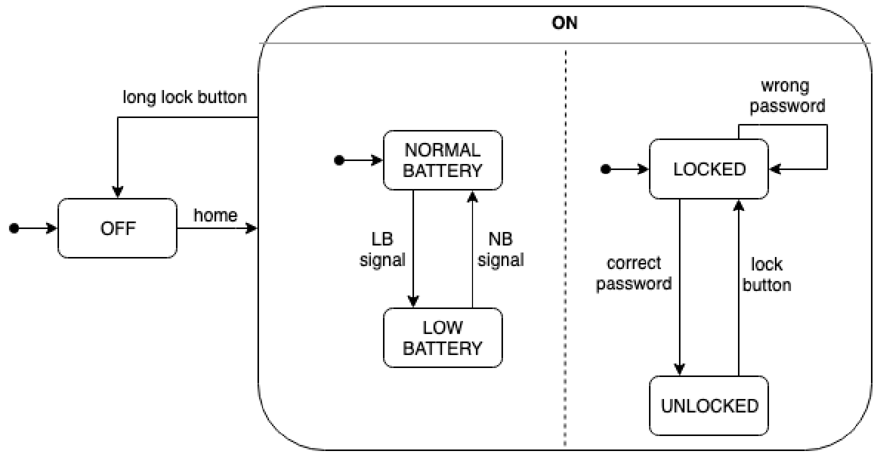
The notation of regions is:



Expanding regions is possible, but highly impractical and usually not wanted, because expanding the region requires creating a state for each combination of states in the different regions. This causes the number of states and transitions to quickly explode. We will not cover how to expand the regions because of this reason.

In general it is best to use small state machine and link these together using super states and regions.

So far when the phone was `ON` we modeled the `LOCKED` and `UNLOCKED` state. When the phone is on, it drains the battery. The system keeps track of the level of the battery. Let's assume that our phone has two battery levels: low battery and normal battery. The draining of the battery and the transitions between the states of this battery runs in parallel to the phone being locked or unlocked. With parallel behavior like this, we can use the regions in our state machine model. the state machine looks like the following, with the new battery states and the regions:



You can see that we assumed the battery to start in the normal level state. Therefore, when the system transitions to the `ON` state it will be in both `LOCKED` and `NORMAL BATTERY` states at once.

Watch our video on Youtube: https://www.youtube.com/embed/D0IQxdjI0M0

Implementing state-based testing in practice

You know a lot about state machines as a model by now. One thing we have not looked at yet is how these state machines are represented in the actual code.

States are very common in programming. Most classes in Object-Oriented-Programming each correspond to their own small state machine.

In these classes, we distinguish two types of methods: **inspection** and **trigger** methods. An **inspection** method only provides information about an object's state. This information consists of the values or fields of an object. The inspection methods only provide information. They do not change the state (or values) of an object. **Trigger** methods bring the class into a new state. This can be done by changing some of the class' values. These trigger methods correspond to the events on the transitions in the state machine.

In a test, we want to bring the class to different states and assert that after each transition, the class is in the expected state. In other words, a **test scenario** is basically a series of calls on the class's trigger methods. Between these calls, we can call the inspection methods to check the state.

Abstraction layer

When a state machine corresponds to a single class, we can easily use the methods described above to test the state machine. However, sometimes the state machine spans over multiple classes. In that case, you might not be able to easily identify the inspection and trigger methods. In this scenario, the state machines can even correspond to *end-to-end testing*. Here, the flow of the entire system is under test, from input to output.

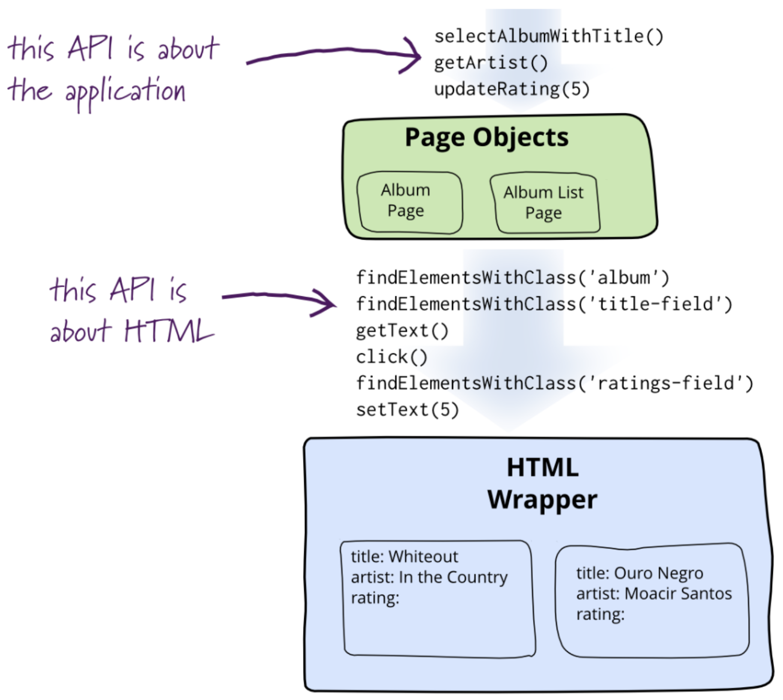
The system under test does not always provide a nice programing interface (API) to inspect the state or trigger the event for the transitions. A common example of such a system is a web application. In the end, the web application works through a browser. To access the state or to trigger the transitions you would then need to use a dedicated tool, like webdriver. Using such a dedicated tool directly is not ideal, as you would have to specify each individual click on the web page to trigger events. What we actually need is an abstraction layer on top of the system under test. This abstraction can just be a Java class, with the methods that we need to be able to test the system as its state machine. Hence, the abstraction layer will contain the inspection method to check the state and the trigger methods to perform the transitions.

With this small abstraction layer, we can formulate the tests very clearly. Triggering a transition is just one method call and checking the state also requires only one method call.

Page Objects

Creating these abstraction layers is very common when testing web applications. In this context, the abstractions are called **Page Objects**.

An example of a page object is shown in the diagram, made by Martin Fowler, below:



At the bottom, you can see a certain web page that we want to test. The tool for communicating through the browser (webdriver for example), gives an API to access the HTML elements. Additionally, the tool supports clicking on elements. For example, on a certain button.

If we use this API directly in the tests, the tests become unreadable very quickly. So, we create a page object with just the methods that we need in the tests. These methods correspond to the application, rather than the HTML elements. The page objects implement these methods by using the API provided by the tool.

Then, the tests use these methods instead of the ones about the HTML elements. Because we are using methods that correspond to the application itself, they will be more readable than tests without the page objects.

State Objects

Page objects give us an abstraction for single pages or even fragments of pages. This is already better than using the API for the HTML elements in the test, but we can take it a bit further. We can make the page objects correspond to the states in the navigational state machine. A navigational state machine is a state machine that describes the flow through a web application. Each page will be a represented as a state. The events of the transitions between these states show how the user can go from one to another page.

With this approach, the page objects each correspond to one of the states of the state machine. Now we do not call them page objects anymore, but **state objects**. In these state objects we have the inspection and trigger methods. Additionally, we have methods that can help with state **self-checking**. These methods verify whether the state itself is working correctly, for example by checking if certain buttons can be clicked on the web page. Now the tests can be expressed in the application's context, using these three types of methods.

Behavior-Driven Design

The state objects are mostly used for end-to-end testing in web development. Another technique useful in end-to-end testing is behavior driven design.

In **behavior driven design** the system is designed with scenario's in mind. These scenario's are written in natural language and describe the system's behavior in a certain situation.

For these scenarios to be used by tools, an example of a tool for scenario's is cucumber, we need to follow a certain format. This is a standard format for scenario's, as it provides a very clear structure. A scenario consists of the following:

- Title of the scenario
- Given ...: Certain conditions that need to hold at the start of the scenario.
- When ...: The action taken.
- Then ...: The result at the end of the scenario.

Let's look at a scenario for an ATM. If we have a balance of $100, a valid card and enough money in the machine, we can give a certain amount of money requested by the user. Together with the money, the card should be given back, and the balance of the account should be decreased. This can be turned into the scenario 1 below:

```
Story: Account Holder withdraws cash

As an Account Holder
I want to withdraw cash from an ATM
So that I can get money when the bank is closed

Scenario 1: Account has sufficient funds
Given the account balance is $100
 And the card is valid
 And the machine contains enough money
When the Account Holder requests $20
Then the ATM should dispense $20
 And the account balance should be $80
 And the card should be returned
```

The small introduction above the scenario itself is part of the user story. A user story usually consists of multiple scenario's with respect to the user introduced. This user is the account holder in this example.

With the general `Given` , `When` , `Then` structure we can describe a state transition as a scenario. In general the scenario for a state transition looks like this:

```
Given I have arrived in some state
When  I trigger a particular event
Then  the application conducts an action
 And  the application moves to some other state.
```

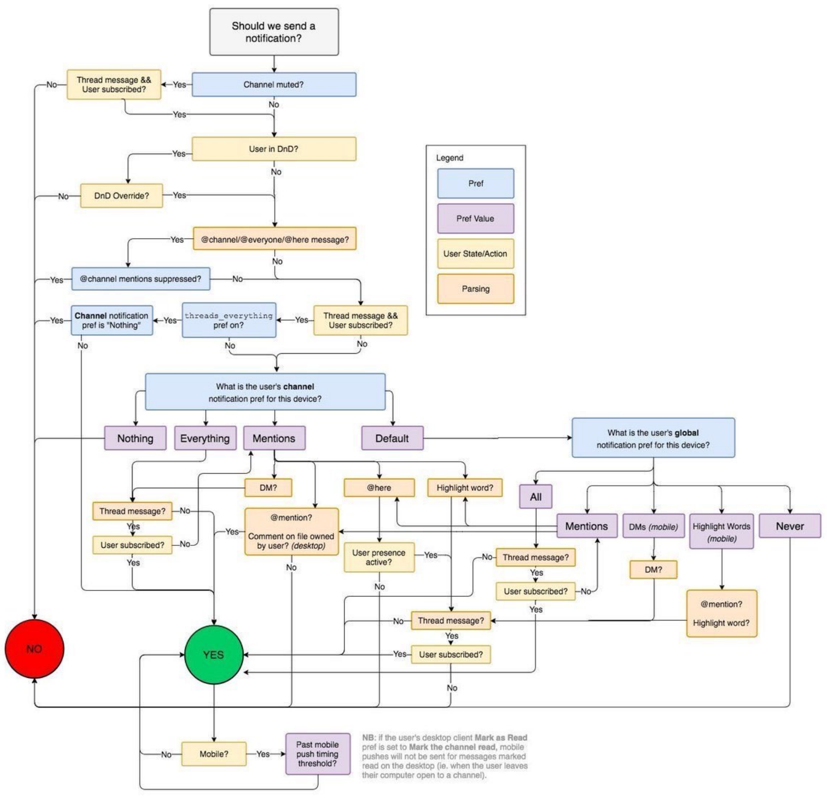
Each scenario will be able to cover only one transition. To get an overview of the system as a whole we will still have to draw the entire state machine.

Watch our video on Youtube:
https://www.youtube.com/embed/NMGX7TEMXdE

Watch our video on Youtube: https://www.youtube.com/embed/gijO3mlcMCg

Other examples of real-world models

Slack shared their internal flow chart that decides whether to send a notification of a message. Impressive, isn't it?


Should we send a notification?
Channel muted?
Thread message && User subscribed?
User in DnD?
DnD Override?
@channel/@everyone/@here message?
@channel mentions suppressed?
Channel notification pref is "Nothing"
threads_everything pref on?
Thread message && User subscribed?
Legend
Pref
Pref Value
User State/Action
Parsing
What is the user's channel notification pref for this device?
Nothing
Everything
Mentions
Default
What is the user's global notification pref for this device?
DM?
@here
Highlight word?
All
Mentions
DMs (mobile)
Highlight Words (mobile)
Never
Thread message?
User subscribed?
@mention? Comment on file owned by user? (desktop)
User presence active?
Thread message?
User subscribed?
DM?
@mention? Highlight word?
Thread message?
User subscribed?
NO
YES
Mobile?
Past mobile push timing threshold?
Yes
No
NB: if the user's desktop client Mark as Read pref is set to Mark the channel read, mobile pushes will not be sent for messages marked read on the desktop (ie. when the user leaves their computer open to a channel).

Exercises

Exercise 1. The *ColdHot* air conditioning system has the following requirements:

- When the user turns it on, the machine is in an *idle* state.
- If it's *too hot*, then, the *cooling* process starts. It goes back to *idle* when the defined *temperature is reached*.
- If it's *too cold*, then, the *heating* process starts. It goes back to *idle* when the defined *temperature is reached*.
- If the user *turns it off*, the machine is *off*. If the user *turns it on* again, the machine is back to *idle*.

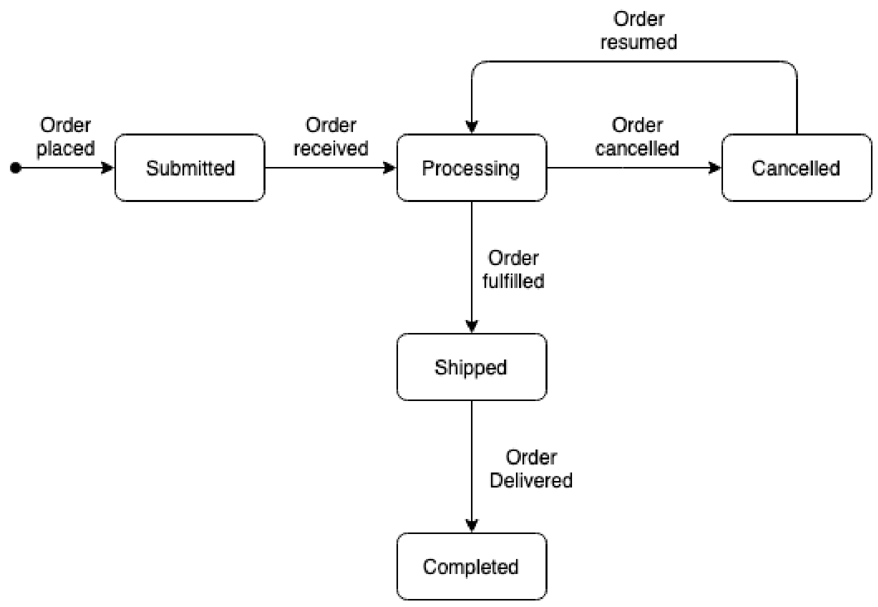
Draw a minimal state machine to represent these requirements.

Exercise 2. Derive the transition tree from the state machine of the assignment above.

Exercise 3. Now derive the transition table of the *ColdHot* state machine.

How many sneaky paths can we test based on the transition table?

Exercise 4. Draw the transition tree of the following state machine:



Use sensible naming for the states in your transition tree.

Exercise 5. With the transition tree you devised in the previous exercise and the state machine in that exercise. What is the transition coverage of a test that the following events: [order placed, order received, order fullfiled, order delivered]?

Exercise 6. Devise the decision table of the state machine that was given in the exercise above. Ignore the initial transition `Order placed`.

Exercise 7. How many sneak paths are there in the state machine we used in the previous exercises? Again ignoring the initial `Order placed` transition.

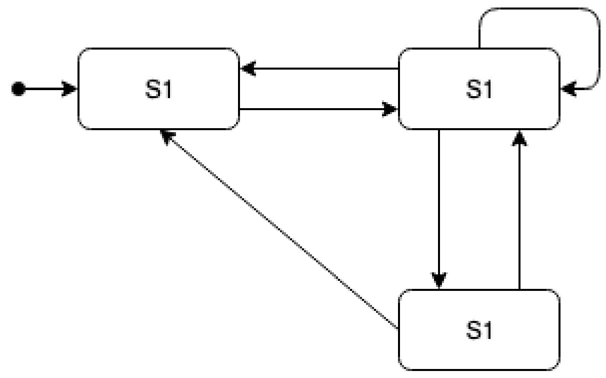
Exercise 8. Consider the following decision table:

Criteria	Options					
C1: Employed for 1 year	T	F	F	T	T	T
C2: Achieved last year's goal	T	dc	dc	F	T	F
C3: Positive evaluation from peers	T	F	T	F	F	T
	10%	0%	5%	2%	6%	3%

Which decision do we have to test for full MC/DC?

Use as few decisions as possible.

Exercise 9. See the following generic state machine.



Draw the transition tree of this state machine.

Exercise 10. The advertisement (ad) feature is an important source of income for the company. Because of that, the life cycle of an ad needs to be better modelled. Our product team defined the following rules:

- The life cycle of an ad starts with an 'empty' ad being created.
- The company provides information about the ad. More specifically, the company defines an image, a description, and how many times it should appear. When all these information is set, the ad then needs to wait for approval.
- An administrator checks the content of the ad. If it follows all the rules, the ad then waits for payment. If the ad contains anything illegal, it then goes back to the very beginning.
- As soon as the company makes the payment, the ad becomes available to users.
- When the number of visualizations is reached, the ad is then considered done. At this moment, the company might consider running the campaign again, which moves the ad to wait for payment again. The company might also decide to simply end the campaign at that moment, which puts the ad in a finalized state.
- While appearing for the users, if more than 10\% of the users complain about the ad, the ad is then marked as blocked. Cute Babies then gets in contact with the company. After understanding the case, the ad either starts to

appear again for the users, or gets marked as innapropriate. An innapropriate ad will never be shown again to the users.

Devise a state diagram that describes the life cycle of an ad.

Exercise 11. A microwave oven has the following requirements:

- Its initial state is `OFF` .
- When the user `turns it on` , the machine goes to an `ON` state.
- If the user selects `warms meal` , then, the `WARMING` process starts. It goes back to `ON` when the defined `time is reached` . A user may `cancel` it at any time, taking the microwave back to the `ON` state.
- If the user selects `defrost meal` , then, the `DEFROSTING` process starts. It goes back to `ON` when the defined `time is reached` . A user may `cancel` it at any time, taking the microwave back to the `ON` state.
- The user can `turn off` the microwave (after which it is `OFF`), but only if the microwave is not warming up or defrosting food.

Draw a minimal state machine to represent the requirements. For this question do not make use of super (OR) states. Also, remember that, if a transition is not specified in the requirements, it simply does not exist, and thus, should not be represented in the state machine.

Exercise 12. Devise a state transition tree for the microwave state machine.

Exercise 13. Again consider the state machine requirements for the microwave. There appears to be some redundancy in the defrosting and warming up functionality, which potentially can be described using super states (also called OR-states). Which effect does this have on the total number of states and transitions for the resulting diagram with a super state?

1. There will be one extra state, and two less transitions.
2. There will be one state less, and the same number of transitions.
3. The total number of states will remain the same, and there will be two less transitions.
4. This has no effect on the total number of states and transitions.

Exercise 14. See the requirement below:

```
Stefan works for Foodgram, a piece of software that enables users to send pict

* The software should only accept images in JPG format.
* The software should not accept images that are bigger than 20MB.
* The software accepts images in both high and low resolution.

As soon as a user uploads a photo, the aforementioned rules are applied.
The software then either says *"Congratulations! Your picture was uploaded suc
```

Create a decision table that takes the three conditions and their respective outcomes into account.

Note: conditions should be modeled as boolean decisions.

Exercise 15 Twitter is a software system that enables users to share short messages within their friends. Twitter's revenue model is ultimately based on advertisements ("ads"). Twitter's system needs to decide when to serve ads to its users, and which ones. For a given user a given ad can be *highly-relevant*, and the system seeks to serve the most relevant ads as often as possible without scaring users away.

To that end, assume that the system employs the following rules to decide whether a user *U* gets served an ad *A* at the moment user *U* opens their Twitter app:

- If the user *U* has not been active during the past two weeks, she will not get to see add *A*;
- If the user *U* has already been served an ad during her last hour of activity, she will not get to see ad *A*;
- Furthermore, if the user *U* has over 1000 followers (an influencer), she will only get to see ad *A* if *A* is labeled as *highly-relevant* for *U*. Otherwise, user *U* will see *A* even if it is not *highly-relevant*.

We can model this procedure in a decision table, in various ways. The complete table would have four conditions and 16 variants. We will try to create a more compact decision table. Note: We will use Boolean conditions only.

One way is to focus on the positive cases only, i.e., specify only the variants in which ad *A* is being served to user *U*. If you don't use 'DC' (don't care) values, how will the decision table look like?

References

- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- van Deursen, A. (2015). Beyond Page Objects: Testing Web Applications with State Objects. ACM Queue, 13(6), 20.

Design by Contracts and Property-Based Testing

Can the production code test itself? In this chapter, we'll discuss what design by contracts and property-based testing are.

Self Testing

A self testing system is, in principle, a system that tests itself. This may sound a bit weird. Let's take a step back first.

The way we tested systems so far was by creating separate classes for the tests. The production code and test code were completely separated. The test suite (consisting of the test classes) exercises and then observes the system under test to check whether it is acting correctly. If the system does something that is not expected by the test suite, the test suite fails. The code in the test suite is completely redundant. It does not add any behavior to the system.

With self-testing, we "move" a bit of the test suite into the system itself. These assertions we insert into production code allows the system to check if it is running correctly by itself. We do not have to run the test suite, but instead the system can check (part of) its behavior during the normal execution of the system. Like with the test suite, if anything is not acting as expected, an error will be thrown. In software testing the self-tests are used as an additional check in the system additional to the test suite.

Assertions

The simplest form of this self-testing is the *assertion*. An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program. In other words, an assertion basically says that a certain condition has to be true at the time the assertion is executed.

In Java, to make an assertion we use the `assert` keyword:

```
assert <condition> : "<message>";
```

The `assert` keywords checks if the `<condition>` is true. If it is, nothing happens. The program just continues its execution as everything is according to plan. However, if the `<condition>` yields false, the `assert` throws an `AssertionError`.

We have implemented a class representing a stack, we just show the `pop` method:

```
public class MyStack {
  public Element pop() {
    assert count() > 0 : " The stack does not have any elements to pop."

    // ... actual method body ...

    assert count() == oldCount - 1;
  }
}
```

In this method, we check if a condition holds at the start: the stack should have at least one element. Then, after the actual method we check whether the count is now one lower than before popping.

These conditions are also known as pre- and postconditions. We cover these in the following section.

Note: The assertion's message is optional, but can be very helpful for debugging. Always include a message that describes what is going wrong if the assertion is failing.

In Java, asserts can be enabled or disabled. If the asserts are disabled, they will never throw an `AssertionError` even if their conditions are false. Java's default configuration is to disable the assertions.

To enable the asserts we have to run Java with a special argument in one of these two ways: `java -enableassertions` or `java -ea`. When using Maven or IntelliJ, the assertions are enabled automatically when running tests. With Eclipse or Gradle, we have to enable it ourselves.

When running the system with assertions enabled, we increase the fault sensitivity of the system, i.e., the system becomes more sensible to faults. This might seem undesirable, but when executing tests, we want the tests to fail. After all, if a test fail, we just found another fault in the system that we can then fix before deploying it to final users.

Watch our video on Youtube:
https://www.youtube.com/embed/Tnm0qwsEiyU

Note how assertions play the role of test oracles here, but in a slightly different way. Just to remember, oracles inform us whether the software behaved correctly.

So far, we only used "value comparison" as oracle. Given an input that we devised, we knew what output to expect. For example, in a program that returns the square root of a number, given n=4, the value we expect as output is 2. Instead of focusing on a specific instances, like we did so far, property checks, like the ones we are going to see in the remaining of this chapter, are more general rules (properties) that we assert on our code. For example, a program that, given n, returns n^2, has a property that it should never return a negative number.

Assertions also serve as an extra safety measure. If it is crucial that a system runs correctly, we can use the asserts to add some additional testing during the system's execution.

However, we note that assertions do not replace unit tests. Assertions are often of a more general nature. We still need the specific cases in the unit tests. A combination of both is what we desire.

Watch our video on Youtube:
https://www.youtube.com/embed/OD0BY8GQs9M

Pre- and Postconditions

We briefly mentioned pre- and postcondition in an example. Now it is time to formalize the idea and see how to create good pre- and postconditions and their influence on the code that we are writing.

Tony Hoare pioneered reasoning about programs with assertions. He proposed the now so-called **Hoare Triples**. A Hoare Triple consists of a set of preconditions $\{P\}$, a program A and a set of postconditions $\{Q\}$ We can express the Hoare Triple as follows: $\{P\}\ A\ \{Q\}$. This can be read as: if we know that P holds, and we execute A, then, we end up in a state where Q holds. If there are no preconditions, i.e., no assumptions needed for the execution of A, we can simply set P to true.

In a Hoare Triple, the A can be a single statement or a whole program. We look at A as a method. Then P and Q are the pre- and postcondition of the method A respectively. Now we can write the Hoare Triple as: $\{preconditions\}\ method\ \{postconditions\}$.

Preconditions

When writing a method, each condition that needs to hold for the method to successfully execute can be a pre-condition. These pre-conditions can be turned into assertions.

Suppose a class that keeps track of our favorite books. Let's define some pre-conditions for the merge method.

The method adds the given books to the list of favorite books and then sends some notification to, e.g., a phone.

```java
public class FavoriteBooks {
  List<Book> favorites;

  // ...

  public void merge(List<Book> books) {
    favorites.addAll(books);
    pushNotification.booksAdded(books);
  }
}
```

When creating pre-conditions, we have to think about what needs to hold to let the method execute. We can focus on the parameter `books` first. This cannot be `null`, because then the `addAll` method will throw a `NullPointerException`. It should also not be empty, because then we cannot add any books. Finally, adding books that we already have does not make sense, so the `books` should not all be present in the `favorites` already.

Now we can take a look at the `favorites`. This also cannot be `null` because then we cannot call `addAll` on favorites.

This gives us 4 pre-conditions and, thus, 4 assertions in the method:

```java
public class FavoriteBooks {
  // ...

  public void merge(List<Book> books) {
    assert books != null;
    assert favorites != null;
    assert !books.isEmpty();
    assert !favorites.containsAll(books);

    favorites.addAll(books);
    pushNotification.booksAdded(books);
  }
}
```

Weakening preconditions

The amount of assumptions made before a method can be executed (and, with that, the amount of pre-conditions) is a design choice.

One might want to weaken the pre-conditions, so that the method accepts/is able to handle more situations. To that aim, we can remove a pre-condition as the method itself can handle the situation where the pre-condition would be false. This makes the method more generally applicable, but is also increases its complexity. The method always has to check some extra things to handle the cases that could had been pre-conditions. Finding the balance between the amount of preconditions and complexity of the method is part of designing the system.

We can remove some of the pre-conditions of the `merge` method by adding some if-statements to the method. First, we can try to remove the `!books.isEmpty()` assertions. This means that the method `merge` has to handle empty `books` lists itself.

```
public class FavoriteBooks {
  // ...

  public void merge(List<Book> books) {
    assert books != null;
    assert favorites != null;
    assert !favorites.containsAll(books);

    if (!books.isEmpty()) {
      favorites.addAll(books);
      pushNotification.booksAdded(books);
    }
  }
}
```

By generalizing the `merge` method, we have removed one of the pre-conditions.

We can even remove the `!favorites.containsAll(books)` assertion by adding some more functionality to the method.

```
public class FavoriteBooks {
  // ...

  public void merge(List<Book> books) {
    assert books != null;
    assert favorites != null;

    List<Book> newBooks = books.removeAll(favorites);

    if (!newBooks.isEmpty()) {
      favorites.addAll(newBooks);
      pushNotification.booksAdded(newBooks);
    }
  }
}
```

Note that, although we increased the complexity of method by removing some of its pre-conditions and dealing with these cases in the implementation, the method now is also easier to be called by clients. After all, the method has less pre-conditions to be called.

Post-conditions

The pre-conditions of a method hold when the method is called. At the end of the method, its **post-conditions** should hold. In other words, the post-conditions formalize the effects that a method guarantees to have when it is done with its execution.

The `merge` method of the previous examples does two things. It adds the new books to the `favorites` list. Let's turn this into a boolean expression, so we can formulate this as a post-condition.

```
public class FavoriteBooks {
  // ...

  public void merge(List<Book> books) {
    assert books != null;
    assert favorites != null;

    List<Book> newBooks = books.removeAll(favorites);

    if (!newBooks.isEmpty()) {
      favorites.addAll(newBooks);
      pushNotification.booksAdded(newBooks);
    }

    assert favorites.containsAll(books);
  }
}
```

The other effect of the method is the notification that is sent. Unfortunately, we cannot easily formalize it as a post-condition. In a test suite, we would probably mock the `pushNotification` and then use `Mockito.verify` to verify that `booksAdded` was called.

It is important to realise that these post-conditions only have to hold if the preconditions held when the method was called. In other words, if the method's pre-conditions were not fully satisfied, the method might not guarantee its post-conditions.

You also saw in the example that we could not really write assertions for some of post-conditions of the method. Post-conditions (and pre-conditions for that matter) might not cover all the possible effects; however, hopefully they do cover a relevant subset of the possible behavior.

Post-conditions of complex methods

For complex methods, the post-conditions also become more complex. That's actually another reason for keeping the method simple. If a method has multiple return statements, it is good to think if these are actually needed. Maybe it is possible to combine them to one return statement with a general post-condition. Otherwise, the post-condition essentially becomes a disjunction of propositions. Each return statement forms a possible post-condition (proposition) and the method guarantees that one of these post-conditions is met.

We have a method body that has three conditions and three different return statements. This also gives us three post-conditions. The placing of these post-conditions now becomes quite important, so the whole method is becoming rather complex with the postconditions.

```
if (A) {
  // ...
  if (B) {
    // ...
    assert PC1;
    return ...;
  } else {
    // ...
    assert PC2;
    return ...;
  }
}
// ...
assert PC3;
return ...;
```

Now if `A` and `B` are true, post-condition 1 should hold. If `A` is true and `B` is false, postcondition 2 should hold. Finally, if `A` is false, postcondition 3 should hold.

How weak pre-conditions affect the post-conditions?

Based on what we saw about pre- and post-conditions, we can come up with a few rules:

- The weaker your pre-condition, the more situations your method is able to handle, and the less thinking your client needs to do. However, with weak pre-conditions, the method will always have to do the checking.
- The post-conditions are only guaranteed if the pre-conditions held; if not, the outcome can any anything. With weak pre-conditions, the method might have to handle different situations, leading to multiple post-conditions guarded by conditions over the inputs or the program state.

Watch our video on Youtube:
https://www.youtube.com/embed/LCJ91VSS3Z8

Invariants

We have seen that preconditions should hold before a method's execution and postconditions should hold after a method's execution. Now we move to conditions that always have to hold, before and after a method's execution. These conditions are called **invariants**. An invariant is thus a condition that holds throughout the entire lifetime of a system, an object, or a data structure.

In terms of implementation, a simple way of using invariants is by creating a "checker" method. This method will go through the data structure/object/system and will assert whether the invariants hold. If an invariant does not hold, it will then throw an `AssertionError`. For simpler invariants, it is common to see a boolean method that checks the invariants of the structure/object/system.

Suppose we have a binary tree datastructure. An invariant for this data structure would be that when a parent points to a child, then the child should point to this parent.

We can make a method that checks if this representation is correct in the given binary tree.

```java
public void checkRep(BinaryTree tree) {
  BinaryTree left = tree.getLeft();
  BinaryTree right = tree.getRight();

  assert (left == null || left.getParent() == tree) &&
      (right == null || right.getParent() == tree)

  if (left != null) {
    checkRep(left);
  }
  if (right != null) {
    checkRep(right);
  }
}
```

In `checkRep()` , we first check if the children nodes of the current node are pointing to this node as parent. Then, we continue by checking the child nodes the same way we checked the current node.

Class invariants

In Object-Oriented Programming, these checks can also be applied at the class-level. This gives us **class invariants**. A class invariant ensures that its conditions will be true throughout the entire lifetime of the object.

The first time a class invariant should be true is right after its construction (i.e., when the constructor method is done). The class invariant should also hold after each public method invocation. Moreover, methods can assume that, when they start, the class invariant holds.

A private method invoked by a public method can leave the object with the class invariant being false. However, the public method that invoked the private method should then fix this and end with the class invariant again being true.

This is all formalized by Bertrand Meyer as: *"The class variant indicates that a proposition P can be a class invariant if it holds after construction, and before and after any call to a public method assuming that the public methods are called with their preconditions being true."*

Of course, we want to know how to implement these class invariants. To implement simple class invariant in Java, we can use the boolean method that checks if the representation is okay. We usually call this method `invariant` . We then assert the return value of this method after the constructor, and before and after each public method. In these public methods, the only pre-conditions and post-conditions that have to hold additionally are the ones that are not in the invariant. When handling more complicated invariants, we can split the invariant into more methods. Then we use these methods in the `invariant` method.

We return to the `FavoriteBooks` with the `merge` method. We had a pre-condition saying that `favorites != null`. Given that this should always be true, we can turn it into a class variant. Additionally, we can add the condition that `pushNotification != null`.

```java
public class FavoriteBooks {
  List<Book> favorites;
  Listener pushNotification;

  public FavoriteBooks(...) {
    favorites = ...;
    pushNotification = ...;

    // ...

    assert invariant();
  }

  protected boolean invariant() {
    return favorites != null && pushNotification != null;
  }

  public void merge(List<Book> books) {
    assert invariant();

    // Remaining preconditions
    assert books != null;

    List<Book> newBooks = books.removeAll(favorites);

    if (!newBooks.isEmpty()) {
      favorites.addAll(newBooks);
      pushNotification.booksAdded(newBooks);
    }

    // Remaining postconditions
    assert favorites.containsAll(books);

    assert invariant();
  }

}
```

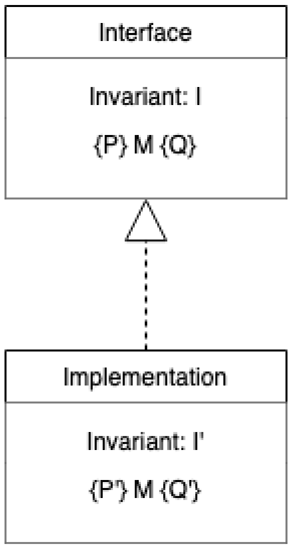
Note that the `invariant` method checks the two conditions. We call `invariant` before and after `merge` and we only assert the pre- and post-conditions that are not covered in the `invariant` method. We also assert the result of the `invariant` method at the end of the constructor.

Watch our video on Youtube:
https://www.youtube.com/embed/T5kwU91W07s

Design by Contracts

Suppose a client system and a server system. The client makes use of the API of of the server. The client and server are bound by a contract. The server does its job as long as its methods are used properly by the client. This relates strongly to the pre- and post-conditions that we discussed earlier. The client has to use the server's methods in a way that their preconditions hold. The server then guarantees that the post-conditions will hold after the method call, i.e., makes sure the method delivers what it promises.

Note how the pre- and post-conditions of the server forms a contract between the server and the client. Designing a system following such contracts is called what we call **Design by Contracts**. In such design, contracts are represented by interfaces. These interfaces are used by the client and implemented by the server. The following UML diagram illustrates it:



Subcontracting

Imagine now an interface. This interface has its own pre- and post-conditions. Now, what happens to these conditions when we create another implementation of this interface (i.e., a class that implements the interface, or a class that extends some base class)?

In the UML diagram above, we see that the implementation can have different pre-, post-conditions, and invariants than its base interface.

In terms of pre-conditions, the new implementation must be able to work with the pre-conditions that were specified in the interface. After all, the interface is the only thing the client sees of the system. The implementation cannot add any pre-conditions to the server's preconditions. In terms of strength, we now know that P' has to be **weaker** than (or as weak as) P.

The postcondition work the other way around. The implementation must do at least the same work as the interface, but it is allowed to do a bit more. Therefore, Q' should be **stronger** than (or as strong as) Q.

Finally, the interface guarantees that the invariant always holds. Then, the implementation should also guarentee that at least the interface's invariant holds.

So, I' should be **stronger** than (or as strong as) I.

In short, using the notation of the UML diagram:

- P' **weaker** than P
- Q' **stronger** than Q
- I' **stronger** than I

The subcontract (the implementation) requires no more and ensures no less than the actual contract (the interface).

Watch our video on Youtube: https://www.youtube.com/embed/aA29jZYdJos

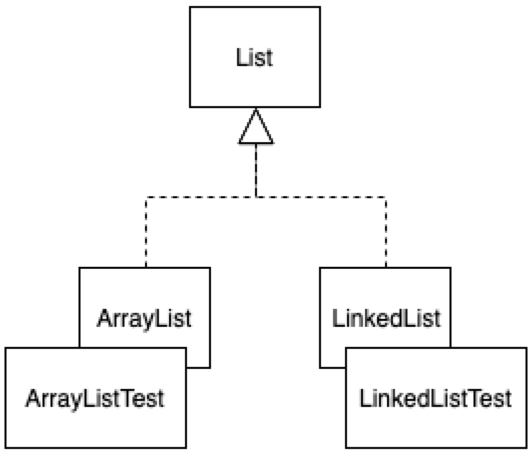
Liskov Substitution Principle

The subcontracting follows the general notion of behavioral subtyping, proposed by Barbara Liskov. The behavioral subtyping states that if we have a class `T` and this class has some sub-classes, the clients or users of class `T` should be able to choose any of `T` 's sub-classes. This notion of behavioral subtyping is now known as the Liskov Substitution Principle (LSP).

In other words, the LSP states that if you use a class, you should be able to replace this class by one of its subclasses. The sub-contracting we discussed earlier is just a formalization of this principle. Proper class hierarchies follow the Liskov Substitution Principle. Leep the LSP in mind when designing and implementating a software system.

How can we test that our classes follow the LSP? To test the LSP, we have to make some test cases for the public methods of the super class and execute these tests with all its subclasses. We could just create the same tests to each of the subclasses' test suites. This, however, leads to a lot of code duplication in the test code, which we would like to avoid.

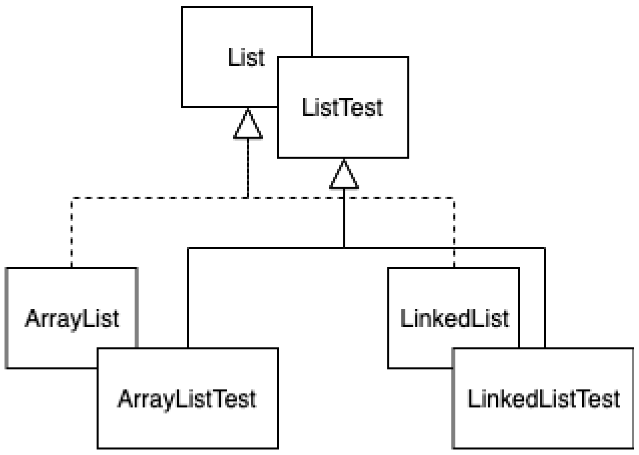
In Java, the List interface is implemented by various sub-classes. Two examples are the `ArrayList` and `LinkedList` . Creating the tests for each of the sub-classes separately will result in the following structure.



The ArrayList and LinkedList will behave the same for the methods defined in List. Therefore, there will be duplicate tests for these methods.

To avoid this code duplication we can create a test suite just for the super class. This test suite tests just the public methods of the super class. The tests in this test suite should then be executed for each of the sub-classes of the super class. This can be done by making the test classes extend the "super test class". The "sub test class" will have all the common tests defined in the "super test class" and its own specific tests.

This is how the test suite looks like:



Here, the `ArrayListTest` and `LinkedListTest` extend the `ListTest`. List is an interface, so the `ListTest` should be abstract. We cannot instantiate a `List` itself, so we should not be able to execute the `ListTest` without a test class corresponding to one of List's subclasses. `ListTest` contains all the common tests of `ArrayListTest` and `LinkedListTest`. `ArrayListTest` and `LinkedListTest` can contain their specific tests.

In the example, you can see that the hierarchy of the test classes is similar to the hierarchy of the classes they test. Therefore, we say that we use a **parallel class hierarchy** in our test classes.

Now we have one problem. How do we make sure that the "super test class" executes its tests in the correct subclass? This depends on the test class that is executed and the subclass it is testing. If we have class `T`, with subclasses `A` and `B`, then when executing tests for `A` we need the test class of `T` to use an instance of `A`; when executing tests for `B`, we need the test class of `T` to use an instance of `B`.

One way to achieve this behavior is by using the *Factory Method* design pattern, which works as follows: In the test class for the interface level (the "super test class"), we define an abstract method that returns an instance with the interface type. By making the method abstract, we force the test classes of the concrete implementations to override it. We return the instance of the specific subclass in the overriden method. This instance is then used to execute the tests.

We want to use the Factory Method design pattern in our tests for the `List`. We start by the interface level test class. Here, we define the abstract method that gives us a List.

```
public abstract class ListTest {

  private List list;

  protected abstract List createList();

  @BeforeEach
  public void setUp() {
    list = createList();
  }

  // Common List tests using list
}
```

Then, for this example, we create a class to test the `ArrayList`. We have to override the createList method and we can define any tests specific for the `ArrayList`.

```
public class ArrayListTest extends ListTest {

  @Override
  protected List createList() {
    return new ArrayList();
  }

  // Tests specific for the ArrayList
}
```

Now, the `ArrayListTest` inherits all the `ListTest`'s tests, so these will be executed when we execute the `ArrayListTest` test suite. Because the `createList()` method returns an `ArrayList`, the common test classes will use an `ArrayList`.

Watch our video on Youtube: https://www.youtube.com/embed/GQ5NTiigkb0

Property-Based Testing

We briefly mentioned property checks. There, we used the properties in assertions. We can also use the properties for test case generation instead of just assertions.

Given that these properties should always hold, we can test them with any input that we like. We typically make use of a generator to try a high number of different inputs, without the need of writing them all ourselves.

These generators often create a series of random input values for a test function. The test function then checks if the property holds using an assertion. For each of the generated input values, this assertion is checked. If we find an input value that makes the assertion to fail, we can affirm that the property does not hold.

The first implementation of this idea was called QuickCheck and was originally developed for Haskell. Nowadays, most languages have an implementation of QuickCheck, including Java. The Java implementation we are going to use is made by Paul Holser and is available on his github. All implementations of QuickCheck follow the same idea, but we will focus on the Java implementation.

How does it work?

- First, we define properties. Similar to defining test methods, we use an annotation on a method with an assertion to define a property: `@Property`. QuickCheck includes a number of generators for various types. For example, Strings, Integers, Lists, Dates, etc.
- To generate values, we can simply add some parameters to the annotated method. The arguments for these parameters will then be automatically generated by QuickCheck. Note that the existing generators are often not enough when we want to test one of our own classes; in these cases, we can create a custom generator which generates random values for this class.

- Quickcheck handles the amount of generated inputs. After all, generating random values for the test input is tricky: the generator might create too much data to efficiently handle while testing.

- Finally, as soon as QuickCheck finds a value that breaks the property, it starts the shrinking process. Using random input values can result in very large inputs. For example lists that are very long or strings with a lot of characters. These inputs can be very hard to debug. Smaller inputs are preferrable when it comes to testing. When an input makes the property fail, QuickCheck tries to find shrink this input while it still makes the property fail. That way it gets the small part of the larger input that actually causes the problem.

As an example: a property of Strings is that if we add two strings together, the length of the result should be the same as the sum of the lengths of the two strings summed. We can use property-based testing and the QuickCheck's implementation to make tests for this property.

```
@Runwith(JUnitQuickcheck.class)
public class PropertyTest {

  @Property
  public void concatenationLength(String s1, String s2) {
    assertEquals(s1.length() + s2.length(), (s1 + s2).length())
  }
}
```

`concatenationLength` had the `Property` anotation, so QuickCheck will generate random values for `s1` and `s2` and execute the test with those values.

Property-based testing changes the way we automate our tests. Usually, we just automate the execution of the tests. With property-based testing, by means of QuickCheck's implementation, we also automatically generate the inputs of the tests.

Property-based testing and AI

A lot of today's research goes into a AI for software testing is about generating good input values for the tests. We try to apply artificial intelligence to find inputs that exercise an important parts of the system.

While the research's results are very promising, there still exist difficulties with this test approach. The main problem is that if the AI generates random inputs, how do we know for sure that the outcome is correct, i.e., how do we know that the problem behaved correctly for that random value? With the unit test we made so far we manually took certain inputs, thought about the correct outcome, and made the assertion to expect that outcome. When generating random inputs, we cannot think about the outcome every time. The pre- and postconditions, contracts, and properties we discussed in this chapter can help us solve the problem. By well-defining the system in these terms, we can use them as oracles in the test cases. The properties always have to be true, so we can use them in all the randomly generated test cases.

We will discuss more about AI techniques in a future chapter.

Watch our video on Youtube:
https://www.youtube.com/embed/7kB6JaSH9p8

Exercises

Exercise 1. See the code below:

```java
public Square squareAt(int x, int y) {
  assert x >= 0;
  assert x < board.getWidth();
  assert y >= 0;
  assert y < board[x].length;
  assert board != null;

  Square result = board[x][y];

  assert result != null;
  return result;
}
```

What assertion(s), if any, can be turned into a class invariant?

Exercise 2. Consider the piece of code in the previous example. Suppose we remove the last assertion (line 10), which states that the result can never be null.

Are the existing preconditions of the `squareAt` method enough to ensure the property in the original line 10? What can we add to the class (other than the just removed postcondition) to guarentee this property?

Exercise 3. Your colleague works on a drawing application. He has created a Rectangle class. For rectangles, the width and height can be different from each other, but can't be negative numbers.

Your colleague also defines the Square class. Squares are a special type of rectangle: the width and height should be equal and also can't be negative. Your colleague decides to implement this by making Square inherit from Rectangle.

The code for the two classes is the following.

```
class Rectangle {
  protected int width;
  protected int height;

  protected boolean invariant() { return width > 0 && height > 0; }

  public Rectangle(int width, int height) {
    assert width > 0;
    assert height > 0;
    this.width = width;
    this.height = height;
    assert invariant()
  }

  public int area() {
    assert invariant();
    int a = width * height;
    assert a > 0;
    return a;
  }

  public void resetSize(int width, int height) {
    assert width > 0;
    assert height > 0;
    this.width = width;
    this.height = height;
    assert invariant();
  }
}

class Square extends Rectangle {
  public Square(int x) {
    assert x > 0;
    super(x, x);
  }

  @Override
  public void resetSize(int width, int height) {
    assert width == height;
    assert width > 0;
    super.resetSize(width, height);
  }
}
```

Inspired by Bertrand Meyer's design by contracts, he also use asserts to make sure contracts are followed. He explicitly defines preconditions and postconditions in various methods of the base Rectangle class and the derived Square class.

A second colleague comes in and expresses concerns about the design. How can you use the assertions provided to discuss the correctness of this design?

Is the second colleagues concern justified? What principle is violated, if any? Explain with the assertions shown in the code.

Exercise 4. You run your application with assertion checking enabled. Unfortunately, it reports an assertion failure signaling a class invariant violation in one of the libraries your application makes use of.

Assume that the contract of the library in question is correct, and that all relevant preconditions are encoded in assertions as well.

Can you fix this problem? If so, how? If not, why?

Exercise 5. HTTP requests return a status code which can have several values. As explained on Wikipedia:

- A 4xx status code "is intended for situations in which the error seems to have been caused by the client". A well known example is the 404 (Page not found) status code.
- A 5xx status code "indicates cases in which the server is aware that it has encountered an error or is otherwise incapable of performing the request." A well known example is the 500 (Internal Server Error) status code.

What is the best correspondence between these status codes and pre- and postconditions?

Exercise 6. A method M belongs to a class C and has a precondition P and a postcondition Q. Now, suppose that a developer creates a class C' that extends C, and creates a method M' that overrides M. Which one of the following statements correctly explains the relative strength of the pre (P') and postconditions (Q') of the overridden method M'?

1. P' should be equal or weaker than P, and Q' should be equal or stronger than Q.
2. P' should be equal or stronger than P, and Q' should be equal or stronger than Q.
3. P' should be equal or weaker than P, and Q' should be equal or weaker than Q.
4. P' should be equal or stronger than P, and Q' should be equal or weaker than Q.

Exercise 7. Which of the following is a valid reason to use assertions in your code?

1. To verify expressions with side effects.
2. To handle exceptional cases in the program.
3. To conduct user input validation.
4. To make debugging easier.

References

- C2 Wiki, What are assertions? http://wiki.c2.com/?WhatAreAssertions
- Mitchell, R., McKim, J., & Meyer, B. (2001). Design by contract, by example. Addison Wesley Longman Publishing Co., Inc..
- Meyer, B. (2002). Design by contract. Prentice Hall.
- Liskov, B. H., & Wing, J. M. (1994). A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6), 1811-1841.
- "Polymorphic Server Test" in Binder, R. V. (1994). Object-oriented software testing. Communications of the ACM, 37(9), 28-30.

The Testing Pyramid

We have studied different techniques to derive test cases. More specifically, specification-based, structural-based, and model-based techniques. However, most of requirements we tested via specification-based techniques had a single responsibility. Most of the source code we tested via structural-based techniques fit in a single unit/class. A large software system, however, is composed of many of those units/responsibilities. Together, they compose the complex behavior of our software system. In this chapter, we are going to discuss the different **test levels** (i.e., unit, integration, and system), their advantages and disadvantages.

Unit testing

In some situatins, our goal is indeed to test a single feature of the software, "purposefully ignoring the other units of the systems". Like we have been done so far. When we test units in isolation, we are doing what we call **unit testing**.

Defining what a 'unit' is, is challenging and highly dependent on the context. A unit can be just one method, or can consist of multiple classes. According to Roy Osherove's unit testing definition: *"A unit test is an automated piece of code that invokes a unit of work in the system. And a unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified."*

As with any testing strategy, unit testing has advantages and disadvantages:

- Firstly, one advantage of unit tests is its speed of execution. A unit test usually takes just a couple of milliseconds to execute. Fast tests give us the ability to test huge portions of the system in a small amount of time.
- Secondly, unit tests are easy to control. We have a high degree of control on how the unit tests exercise the system. A unit test tests the software by giving certain parameters to a method and then comparing the return value of a method to the expected result. The parameters and expected result values are very easy to be adapted/modified in the test.
- Finally, unit tests are easy to write. The JUnit test framework is a very nice and easy framework to write unit tests in. It does not require a lot of setup or additional work to write the tests.

Of course, unit testing also has some disadvantages:

- One of which is the reality of the unit tests. A software system is rarely composed of a single class or fulfills a single logical purpose. The large amount of classes in a system and the interaction between these classes can make the system behave differently in its real application than in the unit tests. Hence, the unit test do not perfectly represent the real execution of a software system.
- Another disadvantage that follows from this is that some bugs simply cannot be caught by means of unit tests. They happen only in the integration of the

different components (which we are not exercising in a pure unit test).

System testing

Unit tests indeed do not exercise the system in a realistic way. To get a more realistic view of the software, we should run it in its entirely. All the components, databases, front end, etc, running together. And then devise tests.

When do it, we are doing what we call **system testing**. In practice, instead of testing small parts of the system in isolation, system tests execute the system as a whole. Note that we can also call system testing as **black-box testing**, because the system is some sort of black box to the tests. We do not care or actually know what goes on inside of the system (the black box) as long as we get the expected output for a given input.

What are the advantages of system testing? The obvious advantage of system testing is how realistic the test are. The system executes as if it is being used in a normal setting. This is good. After all, the more realistic the tests are, the higher the chance of it actually working will be. The system tests also capture the user's perspective more than unit tests do. Faults that the system tests find, can then be fixed before the users would notice the failures that are caused by these faults.

However, system testing also has its downsides. System tests are often slow when compared to unit tests. Having to start and run the whole system, with all its components, takes time. System tests are also harder to write. Some of the components, e.g., databases, require complex setup before they can be used in a testing scenario. This takes additional code that is needed just for automating the tests. Lastly, system tests tend to become flaky. Flaky tests mean that the tests might pass one time, but fail the other time.

Watch our video on Youtube:
https://www.youtube.com/embed/5q_ZjIM_9PQ

Integration testing

Unit and system testing are the two extremes of test levels. Unit tests focus on the smallest parts of the system, while system tests focus on the whole system at once. However, sometimes we want something in between. **Integration testing** offers such a level between unit and system testing.

Let's look at a real example. When a system has an external component, e.g., a database, developers often create a class whose only responsibility is to interact with this external component. Now, instead of testing all the system's components, we just want test this class and its interaction with the component it is made for. One class, one (often external) component. This is integration testing.

In integration testing, we test multiple components of a system, but not the whole system altogether. The tests focus on the interaction between these few components of the system.

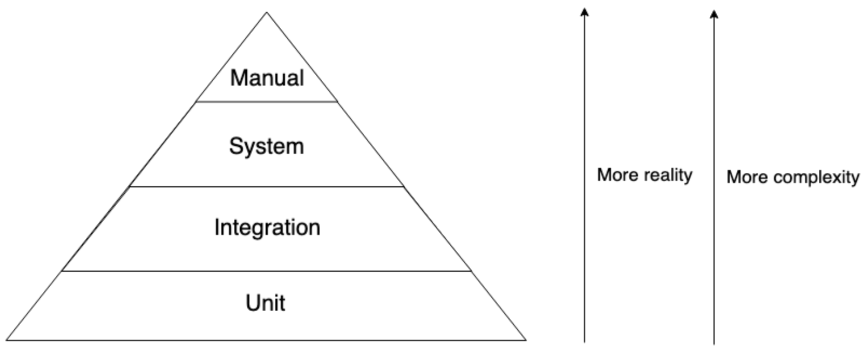
The advantage of this approach is that, while not fully isolated, devising tests just for a specific integration is easier than devising tests for all the components together. Because of that, the effort of writing such tests is a bit higher than when compared to unit tests (and lower when compared to system tests). Setting up the external component, e.g., the database, to the state the tester wants, requires effort.

Can we fully replace system testing with integration testing? After all, it is more real than unit tests and less expensive than system tests. Well... No. Some bugs are really tricky and might only happen in specific sitations where multiple components are working together. There is no silver bullet.

Watch our video on Youtube:
https://www.youtube.com/embed/MPjQXVYqadQ

The Testing Pyramid

We discussed three levels of tests: unit, system, and integration. How much should we do of each? A famous diagram that help us in discussion is the so-called **testing pyramid**.



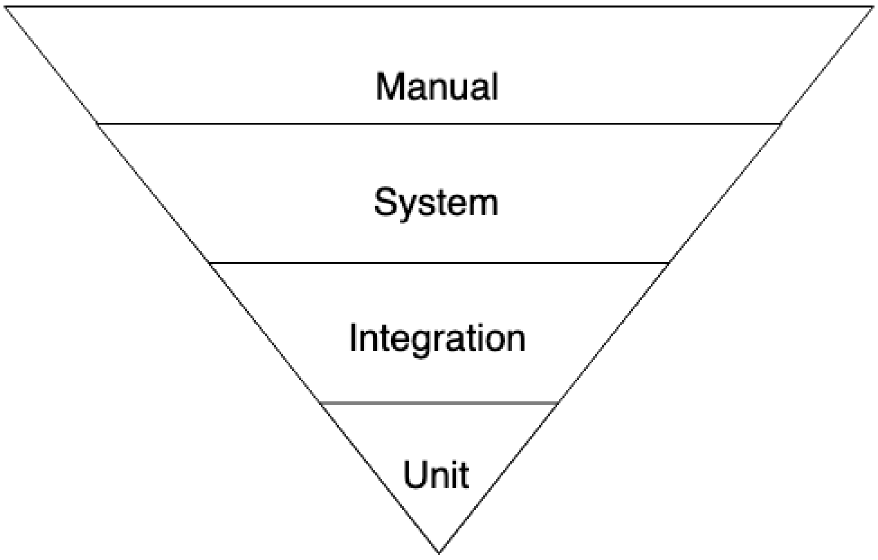
The diagram indicates all the test levels we discussed, plus *manual testing*. The more you go to the top, the more real the test is; however, the more complex it is to devise it.

How much should we do of each then? The common practice in industry is also represented in the diagram. The size of the pyramid slice represents the the amount of test we would want of each test level. Unit test is at the bottom of the pyramid, which means that we want most tests to be unit tests. The reasons for this have been discussed in the unit testing section (they are fast and require less effort to be written). Then, we go for integration tests, of which we do a bit less than unit tests. Given the extra effort that integration tests require, we make sure to write tests for the integrations we indeed need. Lastly, we perform a bit less system tests, and even less manual tests.

Again, note that the further up the pyramid, the closer to reality (and more complex) the tests become. These are important factors in determining what kind of tests to create for a given software system. We will now go over a couple of guidelines you can use to determine what level of tests to write when testing a system (which you should take with a grain of salt; after all, software systems are different from each other, and might require specific guidelines):

- We start at the unit level. When you have implemented a clear algorithm that you want to test, unit tests are the way to go. In an algorithmic piece of code, the parameters are easily controllable, so with unit test you can test the different cases your algorithm might encounter.
- Then, whenever the system is interacting with an external component, e.g. a database or a web service, integration testing is the way to go. For each of the external components that the system uses, integration tests should be created to test the interaction between the system and the external component.
- System tests are very costly, so often we do not test the entire system with it. However, the absolutely critical parts of the system (the ones that should never, ever, stop working), system tests are fundamental.
- As the testing pyramid shows, the manual tests are not done a lot compared to the other testing levels. The main reason is that manual testing is expensive. Manual testing is mainly used for exploratory testing. In practice, exploratory testing helps testers in finding issues that could not be found any other way.

Something you should definitely try to avoid is the so-called *ice cream cone*. The cone is the testing pyramid, but put upside down.



It is common to see teams mostly relying on manual tests. Whenever they have system tests, it is not because they believe on the power of system tests, but because the system was so badly designed, that unit and integration tests are just impossible to be automated. We will discuss design for testability in future chapters.

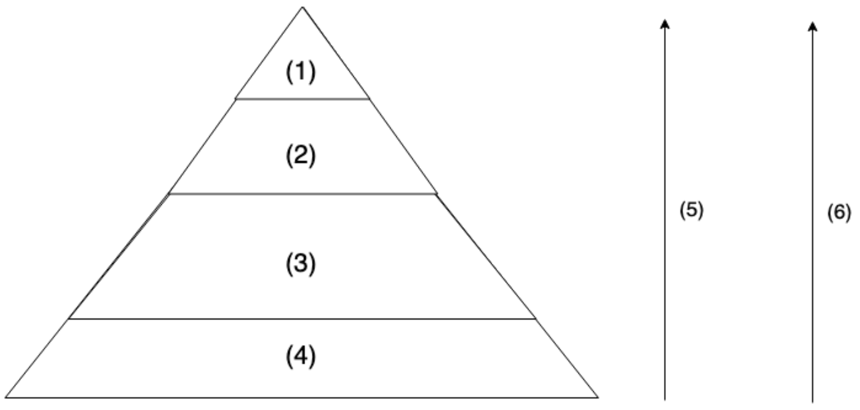
Watch our video on Youtube: https://www.youtube.com/embed/YpKxAicxasU

An academic remark on the testing pyramid

TODO: to be written

Exercises

Exercise 1. Now we have a skeleton for the testing pyramid. What words/sentences should be at the numbers?



(Try to answer the question without scrolling up!)

Exercise 2. As a tester, you have to decide which test level (i.e., unit, integration, or system test) you will apply. Which of the following statements is true?

1. Integration tests, although more complicated (in terms of automation) than unit tests, would better help in finding bugs in the communication with the webservice and/or the communication with the database.
2. Given that unit tests could be easily written (by using mocks) and they would cover as much as integration tests would, it is the best choice in this problem.
3. The most effective way to find bugs in this code is through system tests. In this case, the tester should run the entire system and exercise the batch process. Given that this code can be easily mocked, system tests would also be cheap.
4. While all the test levels can be used for this problem, testers would likely find more bugs if they choose one level and explore all the possibilities and corner cases there.

Exercise 3. The tester should now write an integration test for the `OrderDao` class below.

```java
public class OrderDeliveryBatch {

  public void runBatch() {

    OrderDao dao = new OrderDao();
    DeliveryStartProcess delivery = new DeliveryStartProcess();

    List<Order> orders = dao.paidButNotDelivered();

    for(Order order : orders) {
      delivery.start(order);

      if(order.isInternational()) {
        order.setDeliveryDate("5 days from now");
      } else {
        order.setDeliveryDate("2 days from now");
      }
    }
  }
}

class OrderDao {
  // accesses a database
}

class DeliveryStartProcess {
  // communicates with a third-party webservice
}
```

Which one of the following statements **is not required** when writing an integration test for this class?

1. Reset the database state before each test.
2. Apply the correct schema to the database under test.
3. Assert that any database constraints are met.
4. Set the transaction autocommit to true.

Exercise 4. A newly developed product started off with some basic unit tests but later on decided to only add integration tests and system tests for the new code that was written. This was because a user interacts with the system as a whole and therefore these types of tests were considered more valuable. Thus, unit tests became less prevalent, whereby integration tests and system tests became a more crucial part of the test suite. This transition can be described as:

1. Transitioning from a testing pyramid to an ice-cream cone pattern
2. Transitioning from an ice-cream cone anti-pattern to a testing pyramid
3. Transitioning form an ice-cream cone pattern to a testing pyramid
4. Transitioning from a testing pyramid to an ice-cream cone anti-pattern

Exercise 5. TU Delft just built an in-house software to control the payroll of its employees. The applications makes use of Java web technologies and stores data in a Postgres database. Clearly, the application frequently retrieves, modifies, and inserts large amounts of data into the database. All this communication is made by Java classes that send (complex) SQL queries to the database.

As testers, we know that a bug can be anywhere, including in the SQL queries themselves. We also know that there are many ways to exercise our system. Which one of the following **is not** a good option to detect in the SQL queries?

1. Unit testing.
2. Integration testing.
3. System testing.
4. Stress testing.

Exercise 6. Choosing the level of a test is a matter of a trade-off. After all, each test level has advantages and disadvantages. Which one of the following is the **main advantage** of a test at system level?

1. The interaction with the system is much closer to reality.
2. In a continuous integration environment, system tests provide real feedback to developers.
3. Given that system tests are never flaky, they provide developers with more stable feedback.
4. A system test is written by product owners, making it closer to reality.

Exercise 7. What is the main reason for the number of recommended system tests in the testing pyramid to be smaller than the number of unit tests?

1. Unit tests are as good as system tests.
2. System tests do not provide developers with enough quality feedback.
3. There are no good tools for system tests.
4. System tests tend to be slow and often are non-deterministic.

References

- Chapter 2 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.

- Vocke, Ham. The Practical Test Pyramid (2018), https://martinfowler.com/articles/practical-test-pyramid.html.

- Fowler, Martin. TestingPyramid (2012). https://martinfowler.com/bliki/TestPyramid.html

Mock Objects

When unit testing a piece of code, we want to test it in isolation. However, if the code requires some external dependency to run, e.g., a connection to a database or a webservice, this can be a problem. We do not want to use this external component when we are just testing the piece of code that uses it. How can we do it? We can **simulate the external component**! For that, we will use **mock objects**.

When we mock an object, we create a simulation of this object. To handle external dependencies, we mock (simulate) the class in the system that interacts with the dependency. Instead of doing the actual work of the external components, mock objects just return fake, hard-coded results. These return values can be configured inside of the test itself. We will see how it works in practice soon.

The use of mock objects has some advantages. Returning these pre-configured values is way faster than accessing an external component. Simulating objects also gives us a lot more control. If we, for example, want to make sure the system keeps going even if one of its dependencies crashes, we can just tell the simulation to crash; think of how hard would it be to crash a database just to test if your system reacts to that well (although techniques such as *chaos monkey* have become really popular!).

Mock objects are therefore widely used in software testing, mainly to increase testability. As we have discussed, external systems that the system under test relies on are often mocked to increase controllability and observability. We often mock:

- External components that might be hard to control, or to slow to be used in a unit test, such as databases and webservices.
- To simulate exceptions that are hard to trigger in the actual system.
- To control the behavior of complex third party libraries.
- To test how the different components interact (i.e., exchange messages) among each other.

Implementation-wise, we follow some steps:

- We create a mock object.
- Once it has been created, we give it to the class that normally uses the concrete implementation of the mocked object. This class is now using the mocked object while the tests are executed.
- At first, the mock does not know how to do anything. So, before running the tests, we have to tell it exactly what to do when a certain method is called.
- We trigger the action on the class/method under test. During its execution, note that the mock replaces the external component.
- We make assertions on the mock object, often related to its execution.

Mockito

In Java the most used framework for mocking is Mockito (mockito.org). To perform the steps we mentioned above, we use a couple of methods provided by Mockito:

- `mock(<class>)` : creates a mock object of the given class. The class can be retrieved from any class by `<ClassName>.class` .
- `when(<mock>.<method>).thenReturn(<value>)` : defines the behavior when the given method is called on the mock. In this case `<value>` will be returned.
- `verify(<mock>).<method>` : asserts that the mock object was exercised in the expected way.

Much more functionalities are described in Mockito's documentation.

Suppose we have a method that filters invoices. These invoices are saved in a database and retrieved from the database by this function. The invoices are filtered on their price. All those above 100 are filtered.

```
public class InvoiceFilter {

    public List<Invoice> filter() {

        InvoiceDao invoiceDao = new InvoiceDao();
        List<Invoice> allInvoices = invoiceDao.all();

        List<Invoice> filtered = new ArrayList<>();

        for(Invoice inv : allInvoices) {
            if(inv.getValue() < 100.0)
                filtered.add(inv);
        }

        return filtered;

    }
}
```

Without mocks, the test would need to insert some testing invoices in the database first:

```
@Test
public void filterInvoicesTest() {
  InvoiceDao dao = new InvoiceDao();
  Invoice i1 = new Invoice("M", 20.0);
  Invoice i2 = new Invoice("A", 300.0);
  dao.save(i1);
  dao.save(i2);

  InvoiceFilter f = new InvoiceFilter(dao);
  List<Invoice> result = f.filter();

  assertEquals(1, result.size());
  assertEquals(i1, results.get(0));
  dao.close();
}
```

And of course, clear the database afterwards. Otherwise the test will break in the second run, as there will be two invoices stored in the database! (The database stores data permanenty; so far, we never had to 'clean' the objects; after all, they were always stored in-memory only.)

Watch our video on Youtube:
https://www.youtube.com/embed/0WY7IWbANd8

Now instead of using the database, we want to replace it with a mock object. This way our test will be faster and we can more easily control what the Database-Access-Object returns.

For that to happen, we first need to make sure we can inject the `InvoiceDao` to the `InvoiceFilter` class. Instead of instantiating it, we'll change the `InvoiceFilter` class to actually receive `InvoiceDao` via constructor.

```java
public class InvoiceFilter {

    private InvoiceDao dao;

    public InvoiceFilter (InvoiceDao dao) {
        this.dao = dao;
    }

    public List<Invoice> filter() {

        List<Invoice> allInvoices = dao.all();

        List<Invoice> filtered = new ArrayList<>();

        for(Invoice inv : allInvoices) {
            if(inv.getValue() < 100.0)
                filtered.add(inv);
        }

        return filtered;

    }
}
```

Now, we can mock the `InvoiceDao` class and pass the mocked instance to the `InvoiceFilter`:

```java
@Test
public void filterInvoicesTest() {
  InvoiceDao dao = mock(InvoiceDao.class);
  Invoice i1 = new Invoice("M", 20.0);
  Invoice i2 = new Invoice("A", 300.0);

  List<Invoice> list = Arrays.asList(i1, i2);
  when(dao.all()).thenReturn(list);

  InvoiceFilter f = new InvoiceFilter(dao);
  List<Invoice> result = f.filter();

  assertEquals(1, result.size());
  assertEquals(i1, results.get(0));
}
```

Note how we now have full control over the `InvoiceDao` class. The `InvoiceFilter` uses the `all()` method of the `InvoiceDao` to get the invoices. With the mock, we can easily give the two invoices that we want to test on (note how we are passing `i1` and `i2` . Note also how we do not have to keep a database running while executing the tests!

Watch our video on Youtube:
https://www.youtube.com/embed/kptTWbeLZ3E

Watch our video on Youtube:
https://www.youtube.com/embed/baunKy04deM

More about Mockito

TODO: Add one more example with Mockito here, using the verify()

TODO: Add one more example with Mockito here, using the spy()

Dealing with static methods

TODO: Discuss how to add an abstraction on top of a static method, so that testing becomes easier.

Dealing with APIs you don't control

TODO: Discuss how to add an abstraction on top of things you don't control, e.g., Clock.

Mocking in practice: when to mock?

TODO: Discuss here when to mock

Mocks are a useful tool when it comes to write real isolated unit tests.

Exercises

Exercise 1. See the following class:

```
public class OrderDeliveryBatch {

  public void runBatch() {

    OrderDao dao = new OrderDao();
    DeliveryStartProcess delivery = new DeliveryStartProcess();

    List<Order> orders = dao.paidButNotDelivered();

    for (Order order : orders) {
      delivery.start(order);

      if (order.isInternational()) {
        order.setDeliveryDate("5 days from now");
      } else {
        order.setDeliveryDate("2 days from now");
      }
    }
  }
}

class OrderDao {
  // accesses a database
}

class DeliveryStartProcess {
  // communicates with a third-party webservice
}
```

Which of the following Mockito lines would never appear in a test for the `OrderDeliveryBatch` class?

1. `OrderDao dao = Mockito.mock(OrderDao.class);`
2. `Mockito.verify(delivery).start(order);` (assume `order` is an instance of `Order`)
3. `Mockito.when(dao.paidButNotDelivered()).thenReturn(list);` (assume `dao` is an instance of `OrderDao` and `list` is an instance of `List<Order>`)
4. `OrderDeliveryBatch batch = Mockito.mock(OrderDeliveryBatch.class);`

Exercise 2. You are testing a system that triggers advanced events based on complex combinations of Boolean external conditions relating to the weather (outside temperature, amount of rain, wind, ...). The system has been cleanly designed and consists of a set of cooperating classes that each have a single responsibility. You create a decision table for this logic, and decide to test it using mocks. Which is a valid test strategy?

1. You use mocks to support observing the external conditions.
2. You create mock objects to represent each variant you need to test.

3. You use mocks to control the external conditions and to observe the event being triggered.
4. You use mocks to control the triggered events.

Exercise 3. Below, we show the `InvoiceFilter` class. This class is responsible for returning all the invoices that have an amount smaller than 100.0. It makes use of the InvoiceDAO class, which is responsible for the communication with the database.

```
public class InvoiceFilter {

    private InvoiceDao invoiceDao;

    public InvoiceFilter(InvoiceDao invoiceDao) {
        this.invoiceDao = invoiceDao;
    }

    public List<Invoice> filter() {
        List<Invoice> filtered = new ArrayList<>();
        List<Invoice> allInvoices = invoiceDao.all();

        for(Invoice inv : allInvoices) {
            if(inv.getValue() < 100.0)
                filtered.add(inv);
        }

        return filtered;
    }
}
```

Which of the following statements is **false** about this class?

1. Integration tests would help us achieve a 100% branch coverage, which is not possible solely via unit tests.
2. Its implementation allows for dependency injection, which enables mocking.
3. It is possible to write completely isolated unit tests for it by, e.g., using mocks.
4. The InvoiceDao class (a direct dependency of the InvoiceFilter) itself should be tested by means of integration tests.

Exercise 4. Class A depends on a static method in another class B. Suppose you want to test class A, which approach(es) can you take to be able to test properly?

1. Mock class B to control the behavior of the methods in class B.
2. Refactor class A, so the outcome of the method of class B is now used as an parameter.

3. Only approach 1.

4. Neither.
5. Only approach 2.
6. Both.

References

- Fowler, Martin. Mocks aren't stubs. https://martinfowler.com/articles/mocksArentStubs.html

- Mockito's website: https://site.mockito.org

Design for Testability

Introduction

Now that we know how to use mocks and what we can use them for, it is time to take another look at how we design our software systems.

In the example of the Mocking chapter, it was very easy to make sure that the `InvoiceFilter` used our mock, instead of the concrete instance of the class. This is not always the case. Note how we had to refactor the code for that happen. **We have to design our code in a way that it makes it easy to test the code!**

Testability is related to the easeness of writing automated tests to that code. We know that automated tests are crucial in software development; therefore, it is essential that our code is testable.

In this chapter, we'll discuss some design practices that increases the testability of our software systems.

Watch our video on Youtube: https://www.youtube.com/embed/iVJNaG3iqrQ

Dependency Injection

Dependency injection is a design choice we can use to make our code more testable. We will illustrate what dependency injection is it by means of an analogy:

Say we need a hammer to perform a certain task. When we are asked to do this task, we could go search and find the hammer. Then, once we have the hammer, we can use it to perform the task. However, another approach is to say that we need a hammer when someone asks us to perform the task. This way, instead of getting the hammer ourselves, we get it from the person that wants us to do the task.

We can do the same when managing the dependencies in our code. Instead of the class instantiating dependency itself, the class asks for the dependency (via constructor or a setter, for example).

We actually applied this idea in the Mocking chapter already. But let's revisit it: In the Mockito example, it was easy to make the `InvoiveFilter` use the mock. Let's assume that the `InvoiveFilter` is implemented as follows:

```
public class InvoiceFilter {

  public void filter() {
    InvoiceDao dao = new InvoiceDao();

    // ...
  }

}
```

In other words, the `InvoiceFilter` itself instantiates the `InvoiceDao` . In this implementation, there is no way to pass the mock to the `InvoiceFilter` . Any test code we write will necessarily use a concrete `InvoiceDao` . As we cannot control the way the `InvoiceDao` operates, this makes it a lot harder to write automated tests. Our tests will need a working database to run!

Instead, we can design our class in a way that it allows dependencies to be injected. Note how we receive the dependency via constructor now:

```
public class InvoiceFilter {

  private InvoiceDao invoiceDao;

  public InvoiceFilter(InvoiceDao invoiceDao) {
    this.invoiceDao = invoiceDao;
  }

  public void filter() {
    // ...
  }
}
```

Now we can instantiate the `InvoiceFilter` and pass it a `InvoiceDao` mock in the test code. This `InvoiceFilter` also enables the production code to pass a concrete `InvoiceDao` when the program is run normally (after all, we want it to access the real database in production!).

This simple change in the design of the class makes the creation of automated tests easier and, therefore, increases the testability of the code.

The dependency injection principle improves our code in many ways:

- Enables us to mock the dependencies in the test code.
- Makes all the dependencies more explicit; after all, they all need to be injected (via constructor, for example).
- The class becomes more extensible. As a client of the class, you can pass any dependency via the constructor. Suppose a class depends on a type `A` (and receives it via constructor). As a client, you can pass `A` or any implementation of `A` , e.g., if `A` is `List` , you can pass `ArrayList` or `LinkedList` . Your class now can work with many different implementations of `A` . (You should read more about the Open/Closed Principle).

Watch our video on Youtube:
https://www.youtube.com/embed/mGdsdBEWB5E

Ports and Adapters

The way a software system is designed influences how easy it is to test the system. As we discussed before, the idea of designing a system in a way that it is testable is called **design for testability**. It is important that a software developer knows about certain ways to design for testability in order to implement systems well.

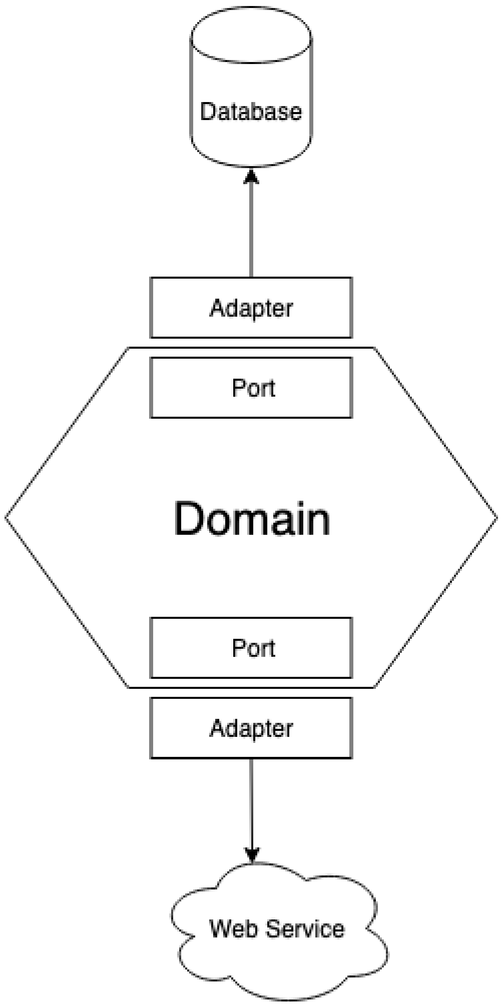
In general, design for testability comes down to separating the domain and the infrastructure of the systems. The domain is the core of the system: The business rules, its logic, its entities, etc. This is very specific to the system that is being built.

A system also commonly makes use of a database, or webservices, or any other external components. We consider them as *infrastructure*. In our software systems, we often have classes which the role is to make a bridge between the external component and the core of our sytem.

To increase testability, we need to separate these two layers (core/domain and infrastructure) as much as possible. In practice, we observe that when domain code and infrastructure code are mixed up together, the system just gets harder to test.

To separate the domain and the infrastructure, we can use the idea of **Ports and Adapters**, as named by Alistair Cockburn. Ports & Adapters is also called the **Hexagonal Architecture**. Following this idea, the domain (business logic) depends on "Ports" rather than directly on the infrastructure. These ports are just interfaces that define what the infrastructure is able to do. These ports are completely separated from the implementation of the infrastructure. The "adapters", on the other hand, are very close to the infrastructure. These are the implementations of the ports that talk to the database or webservice etc. They know how the infrastructure works and how to communicate with it.

In the schema below, you can see that the ports are therefore part of the domain.



Ports and Adapters helps us a lot with the testability of our code. After all, if our core domain depends only on ports, we can easily mock them!

To sum up, make sure your domain/core code does not depend on the infrastructure directly. Create a layer that abstracts the infrastructure away! This layer can then be easily mocked.

Note: The idea of separating the domain and the infrastructure is also intensively discussed in the Domain-Driven Design book (which we recommend you to read!).

Watch our video on Youtube: https://www.youtube.com/embed/hv1XV87lJgA

Design for testability tips

We end this chapter with a couple of practical tips that you can use to create a well designed, testable software system.

- **Cohesion**: Cohesive classes are classes that do only one thing. Cohesive classes tend to be easier to test. If one of your classes is not cohesive, you can split the class into multiple small cohesive classes. This way you can test each separate class more easily.
- **Coupling**: Coupling represents the amount of classes that a class depends on. A highly coupled class needs a lot of other classes to work. Coupling hurts testability. Imagine having to create ten mocks, define the behavior of these ten mocks just one class. Reducing coupling is tricky and maybe one of the biggest challenges in software design. To reduce coupling, you can, for example, group some of the dependencies together and/or create bigger abstractions.
- **Complex Conditions**: When conditions are very complex (i.e., are composed of multiple boolean operations), maybe it is better split them into multiple conditions. Testing these simpler conditions is easier than testing one very complex condition.
- **Private methods**: A common question is whether to test private methods or not. The answer is usually "no". In principle, we would like to test the private methods only through the public methods that use them. This means that we test if the public method behaves correctly, not the private method. If you feel like you want to test the private method in isolation (because it might be a very large important method), this might mean that the method should probably have its own separate class, which can be tested in isolation.
- **Static metods**: Static methods can hurt testability, as we cannot control the behavior of the static methods (i.e., they are hard to mock). Therefore, we avoid using static methods whenever possible.
- **Layers**: When using other's code or external dependencies, creating layers/classes that abstract away the dependency might help you in increasing testability. Do not be afraid to create these extra layers (although they do add complexity).
- **Infrastructure**: Again, make sure not to mix the infrastructure with the domain. You can use the methods we discussed above.
- **Encapsulation**: Encapsulation is about bringing the behavior of a class close to the data. Encapsulation will help you identify the problems in the code easier.

Note how there is a deep synergy between well design production code and testability!

Watch our video on Youtube:
https://www.youtube.com/embed/VaScxLhsDBQ

Exercises

Exercise 1. How can we improve the testability of the `OrderDeliveryBatch` class?

```
public class OrderDeliveryBatch {

  public void runBatch() {

    OrderDao dao = new OrderDao();
    DeliveryStartProcess delivery = new DeliveryStartProcess();

    List<Order> orders = dao.paidButNotDelivered();

    for (Order order : orders) {
      delivery.start(order);

      if (order.isInternational()) {
        order.setDeliveryDate("5 days from now");
      } else {
        order.setDeliveryDate("2 days from now");
      }
    }
  }
}

class OrderDao {
  // accesses a database
}

class DeliveryStartProcess {
  // communicates with a third-party webservice
}
```

What techniques can we apply? What would the new implementation look like? Think about what you would need to do to test the `OrderDeliveryBatch` class.

Exercise 2. Consider the following requirement and implementation.

```
A webshop gives a discount of 15% whenever it's King's Day.
```

```
public class KingsDayDiscount {

  public double discount(double value) {

    Calendar today = Calendar.getInstance();

    boolean isKingsDay = today.get(MONTH) == Calendar.APRIL
        && today.get(DAY_OF_MONTH) == 27;

    return isKingsDay ? value * 0.15 : 0;

  }
}
```

We want to create a unit test for this class.

Why does this class have bad testability? What can we do to improve the testability?

I.e. what makes it very hard to test the method?

Exercise 3. Sarah just joined a mobile app team that has been trying to write automated tests for a while. The team wants to write unit tests for some part of their code, but "that's really hard", according to the developers.

After some code review, the developers themselves listed the following problems in their codebase:

1. May classes mix infrastructure and business rules
2. The database has large tables and no indexes
3. Use of static methods
4. Some classes have too many attributes/fields

To increase the testability, the team has budget to work on two out of the four issues above. Which items should Sarah recommend them to tackle first?

Note: All of the four issues should obviously be fixed. However, try to prioritize the two most important onces: Which influence the testability the most?

Exercise 4. Observability and controllability are two important concepts when it comes to software testing. The three developers below could benefit from improving either the observability or the controllability of the system/class under test.

1. "I can't really assert that the method under test worked well."
2. "I need to make sure this class starts with that boolean set to false, but I simply can't do it."
3. "I just instantiated the mock object, but there's simply no way to inject it in the class."

For each of the problems above: is it related to observability or controllability?

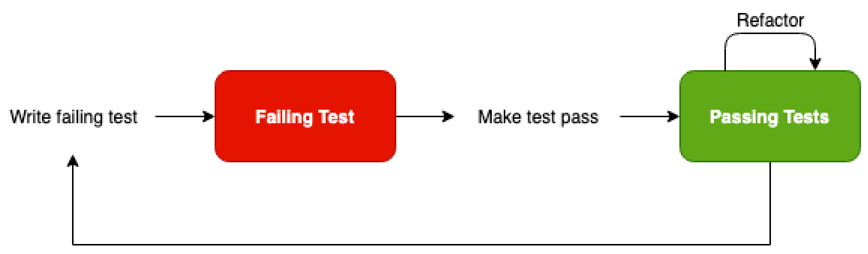
References

- Cockburn, Alistair. The Hexagonal Architecture. https://wiki.c2.com/?HexagonalArchitecture
- Hevery, Misko. The Testability Guide. http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf
- Michael Feathers. The deep synergy between well design production code and testability. https://www.youtube.com/watch?v=4cVZvoFGJTU

Test-Driven Development

We are used to write some production code and, once we are finished, we move to writing the tests. One disadvantage of this approach is that we might create the tests only much later. Test-Driven Development (TDD) suggests the opposite: why don't we start by writing the tests and only then production code?

The TDD cycle is illustrated in the diagram below.



The first think of the test. Then we write the test. This test will fail, because we have not written the code yet. Now that we have a failing test, we write code that makes the test pass.

And we do all of this in the simplest way we can. This simplicity means that we create the simplest implementation that solves the problem and we start with the simplest possible test case. After we wrote the code that makes the test pass, it is time to refactor the code we just made.

TODO: Missing a video about how TDD works here

TDD has some advantages:

- By creating the test first, we also look at the requirements first. This makes us write the code for the specific problem that it is supposed to solve. In its turn, this prevents us from writing useless, unnecessary code.
- We can control our pace of writing the production code. Once we have a failing test, our goal is clear: To make the test pass. With the test that we create, we can control the pace in which we write the production code. If we are confident in how to solve the problem, we can make a big step by creating a complicated test. However, if we are not sure how to tackle the problem, we can break it into small parts and start with the simpler pieces first by creating tests for those pieces.
- The tests are derived from the requirements. Therefore, we know that, when the tests pass, the code does what it is supposed to do. When writing tests from the source code instead, the tests might pass while the code is not doing the right thing. The tests also show us how easy it is to use the class we just made. We are using the class directly in the tests so we know immediately when we can improve something.

- Related to the previous point is the way we design the code when using TDD. Creating the tests first makes us think about the way to test the classes before implementing them. It changes the way we design the code. This is why Test Driven Development is sometimes also called Test Driven Design. Thinking about this design earlier is also better, because it is easier to change the design of a class early on. You probably have to change less if you just start with a class than when the class is utilized by other parts of the system. We discuss more about Test-Driven Design later.
- Writing tests first gives faster feedback on the code that we are writing. Instead of writing a lot of code and then a lot of tests, i.e. getting a lot of feedback at once after a long period of time, we create a test and then write a small piece of code for that test. Now we get feedback after each test that we write, which is usually after a piece of code that is much smaller than the pieces of code we test at once in the traditional approach. Whenever we have a problem it will be easier to identify it, as the added code is smaller. Moreover if we do not like anything it is easier to improve the code.
- We already talked about the importance of controllability when creating automated tests. By thinking about the tests before implementing the code we automatically make sure that the code is easy to control. This improves the code quality as it will be easier to test the controllable code.
- Finally, the tests that we write can indicate some problems in the code. Too many tests for just one class can indicate that the class has too many functionalities and that it should be split up into more classes. If we need too many mocks inside of the tests the class might be too coupled, i.e. it needs too many other classes to function. If it is very complex to set everything up for the test, we may have to think about the pre-conditions that the class uses. Maybe there are too many or maybe some are not necessary.

Given all these advantages, should we use TDD 100% of time? There is, of course, no universal answer. While some research shows the advantages of TDD, others are more in doubt about it.

TODO: Missing a video about the advantages of TDD here

Test-Driven Design

Will I create a better class design if I use Test-Driven Development? Well, yes and no; TDD doesn't do magic. In the following, we discuss the effects of the practice, and how the practice can help developers during class design.

Developers commonly argue that TDD helps software design, improving internal code quality. For example, Kent Beck [1], Robert Martin [2], Steve Freeman [3], and Dave Astels [4], state in their books (without scientific evidence) that the writing of unit tests in a TDD fashion promotes significant improvement in class design, helping developers to create simpler, more cohesive, and less coupled classes. They even consider TDD as a class design technique [5], [6]. Nevertheless, the way in which TDD actually guides developers during class design is not yet clear.

After interviewing some developers, we observed the following.

TDD does not drive to a better design by itself

Participants affirmed that the practice of TDD did not change their class design during the experiment. The main justification was that their experience and previous knowledge regarding object-orientation and design principles guided them during class design. They also affirmed that a developer with no knowledge in object-oriented design would not create a good class design just by practicing TDD or writing unit tests.

The participants gave two good examples reinforcing the point. One of them said that he made use of a design pattern he learned a few days before. Another participant mentioned that his studies on SOLID principles helped him during the exercises.

The only participant who had never practiced TDD before stated that he did not feel any improvement in the class design when practicing the technique. Curiously, this participant said that he considered TDD a design technique. It somehow indicates that the popularity of the effects of TDD in class design is high. That opinion was slightly different from that of experienced participants, who affirmed that TDD was not only about design, but also about testing.

However, different from the idea that TDD and unit tests do not guide developers directly to a good class design, all participants said that TDD has positive effects on class design. Many of them mentioned the difficulty of trying to stop using TDD or thinking about tests, what can be one reason for not having significant difference in terms of design quality in the code produced with and without TDD.

"When you are about to implement something, you end up thinking about the tests that you'll do. It is hard to think "write code without thinking about tests!". As soon as you get used to it, you just don't know another way to write code..."

According to them, TDD can help during class design process, but in order to achieve that, the developer should have certain experience in software development. Most participants affirmed that their class designs were based on their experiences and past learning processes. In their opinion, the best option is to link the practice of TDD and experience.

Baby steps and simplicity

TDD suggests developers to work in small (baby) steps; one should define the smallest possible functionality, write the simplest code that makes the test green, and do one refactoring at a time.

In the interviews, participants commented about this. One of them mentioned that, when not writing tests, a developer thinks about the class design at once, creating a more complex structure than needed.

One of the participants clearly stated how he makes use of baby steps, and how it helps him think better about his class design:

“Because we start to think of the small and not the whole. When I do TDD, I write a simple rule (...), and then I write the method. If the test passes, it passes! As you go step by step, the architecture gets nice. (...) I used to think about the whole (...). I think our brain works better when you think small. If you think big, it is clear, at least for me, that you will end up forgetting something. ”

Refactoring confidence

Participants affirmed that, during the process of class design, changing minds is constant. After all, there is still a small knowledge about the problem, and about how the class should be built. This was the most mentioned point by the participants. According to them, an intrinsic advantage of TDD is the generated test suite. It allows developers to change their minds and refactor all the class design safely. Confidence, according to them, is an important factor when changing class design or even implementation.

"It gives me the opportunity to learn along the way and make things differently. (…). The test gives you confidence."

A participant even mentioned a real experience, in which TDD made the difference. According to him, he changed his mind about the class design many times and trusted the test suite to guarantee the expected behavior.

A safe space to think

In an analogy done by one of the participants, tests are like draft paper, in which they can try different approaches and change their minds about it frequently. According to them, when starting by the test, developers are, for the first time, using their own class. It makes developers look for a better and clearer way to invoke the class’ behaviors, and facilitate its use:

"Tests help you on that. They are a draft for you to try to model it the best way you can. If you had to model the class only once, it is like if you have only one chance. Or if you make it wrong, fixing it would give you a lot of work. The idea of you having tests and thinking about them, it is like if you have an empty draft sheet, in which you can put and remove stuff because that stuff doesn’t even exist."

We asked their reasons for not thinking on the class design even when they were not practicing TDD or writing tests. According to them, when a developer does not practice TDD, they get too focused on the code they are writing, and thus, they end up not thinking about the class design they were creating. They believe tests make them think of how the class being created will interact with the other classes and of the easiness of using it.

One of the participants was even more precise in his statement. According to him, developers that do not practice TDD, as they do not think about the class design they are building, they end up not doing good use of OOP. TDD forces developers to speed down, allowing them to think better about what they are doing.

Rapid feedback

Participants also commented that one difference they perceived when they practiced TDD was the constant feedback. In traditional testing, the time between the production code writing and test code writing is too long. When practicing TDD, developers are forced to write the test first, and thus receive the feedback a test can provide sooner.

One participant commented that, from the test, developers observe and criticize the code they are designing. As the tests are done constantly, developers end up continuously thinking about the code and its quality:

"When you write the test, you soon perceive what you don't like in it (...). You don't perceive that until you start using tests."

Reducing the time between the code writing and test writing also helps developers to create code that effectively solves the problem. According to the participants, in traditional testing, developers write too much code before actually knowing if it works.

The search for testability

Maybe the main reason for the practice of TDD helping developers in their class design is the constant search for testability. It is possible to infer that, when starting to produce code by its test, the production code should be, necessarily, testable.

When it is not easy to test a specific piece of code, developers understand it as a class design smell. When this happens, developers usually try to refactor the code to make it easier to test. A participant also affirmed that he takes it as a rule; if it is hard to test, then the code may be improved.

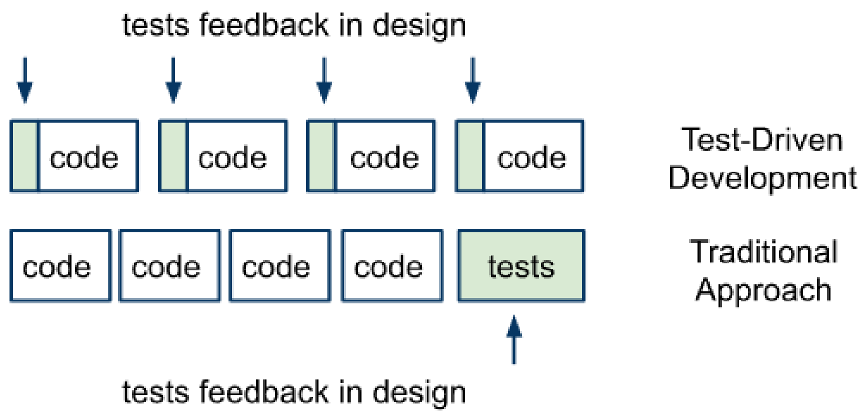
Feathers [7] raised this point: the harder it is to write the test, the higher the chance of a class design problem. According to him, there is a strong synergy between a highly testable class and a good class design; if developers are looking for testability, they end up creating good class design; if they are looking for good class design, they end up writing testable code.

Participants went even further. During the interviews, many of them mentioned patterns that made (and make) them think about possible design problems in the class they build. As an example, they told us that when they feel the need to write many different unit tests to a single method, this may be a sign of a non-cohesive method. They also said that when a developer feels the need to write a big scenario for a single class or method, it is possible to infer that this need emerges in classes dealing with too many objects or containing too many responsibilities, and thus, it is not cohesive. They also mentioned how they detect coupling issues. According to them, the abusive use of mocking objects indicates that the class under testing has coupling issues.

What did we learn from it?

The first interesting myth contested by the participants was the idea that the practice of TDD would drive developers towards a better design by itself. As they explained, the previous experience and knowledge in good design is what makes the difference; however, TDD helps developers by giving feedback by means of the unit tests that they are writing constantly. As they also mentioned, the search for testability also makes them rethink about the class design many times during the day—if a class is not easy to be tested, then they refactor it.

We agree with the rationale. In fact, when comparing to test-last approaches, developers do not have the constant feedback or the need to write testable code. They will have the same feedback only at the end, when all the production code is already written. That may be too late (or expensive) to make a big refactoring in the class design.



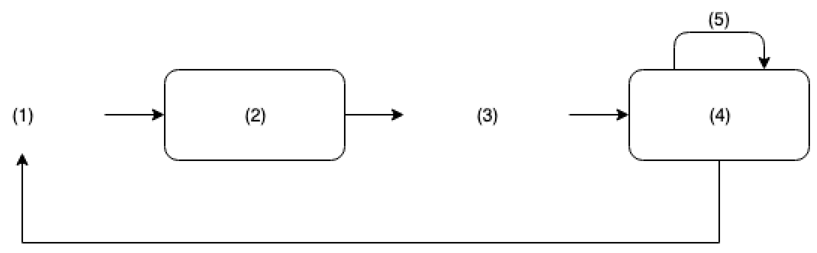
We also agree with the confidence when refactoring. As TDD forces developers to write unit tests frequently, those tests end up working as a safety net. Any broken change in the code is quickly identified. This safety net makes developers more confident to try and experiment new design changes—after all, if the changes break anything, tests will warn developers about it. That is why they also believe the tests are a safe space to think.

Therefore, we believe that is is not the practice by itself that helps developers to improve their class design; but it is the consequences of the constant act of writing a unit test, make that class testable, and refactor the code, that drives developers through a better design.

Conclusion: Developers believe that the practice of test-driven development helps them to improve their class design, as the constant need of writing a unit test for each piece of the software forces them to create testable classes. These small feedbacks—is your test easy to be tested or not?—makes them think and rethink about the class and improve it. Also, as the number of tests grow, they act as a safety net, allowing developers to refactor freely, and also try and experiment different approaches to that code. Based on that, we suggest developers to experiment the practice of test-driven development, as its effects look positive to software developers.

Exercises

Exercise 1. We have the following skeleton for a diagram illustrating the Test Driven Development cycle. What words/sentences should be at the numbers?



(Try to answer the question without scrolling up!)

Exercise 2. Remember the `RomanNumeral` problem?

```
**The Roman Numeral problem**

It is our goal to implement a program that receives a string as a parameter
containing a roman number and then converts it to an integer.

In roman numeral, letters represent values:

* I = 1
* V = 5
* X = 10
* L = 50
* C = 100
* D = 500
* M = 1000

We can combine these letters to form numbers.
The letters should be ordered from the highest to the lowest value.
For example `CCXVI` would be 216.

When we put a lower value in front of a higher one, we substract that value fr
For example we make 40 not by XXXX, but instead we use {% math %}50 - 10 = 40{
Combining both these principles we could give our method `MDCCCXLII` and it sh
```

Implement this program. Practice TDD!

Exercise 3. Which of the following **is the least important** reason why one does Test-Driven Development?

1. As a consequence of the practice of TDD, software systems get completely tested.
2. TDD practitioners use the feedback from the test code as a design hint.
3. The practice of TDD enables developers to have steady, incremental progress throughout the development of a feature.
4. The use of mocks objects helps developers to understand the relationships between objects.

Exercise 4. TDD has become a really popular practice among developers. According to them, TDD has several benefits. Which of the following statements **is not** considered a benefit from the practice of TDD?

Note: We are looking at the perspective of developers, which may not always match the results of empirical research.

1. Better team integration. Writing tests is a social activity and makes the team more aware of their code quality.
2. Baby steps. Developers can take smaller steps whenever they feel they have to.
3. Refactoring. The cycle suggests developers to constantly improve their code.
4. Design for testability. Developers are "forced" to write testable code from the very beginning.

References

- Beck, K. (2003). Test-driven development: by example. Addison-Wesley Professional.
- [1] Beck K (2002) Test-driven development by example. 1° edn. Addison-Wesley Professional, Boston, USA.
- [2] Martin R (2006) Agile principles, patterns, and practices in C#. 1st edition. Prentice Hall, Upper Saddle River.
- [3] Steve Freeman, Nat Pryce (2009) Growing object-oriented software, Guided by Tests. 1° edn. Addison-Wesley Professional, Boston, USA.
- [4] Astels D (2003) Test-driven development: a practical guide. 2nd edition. Prentice Hall.
- [5] Janzen D, Saiedian H (2005) Test-driven development concepts, taxonomy, and future direction. Computer 38(9): 43–50. doi:10.1109/MC.2005.314.
- [6] Beck K (2001) Aim, fire. IEEE Softw 18: 87–89. doi:10.1109/52.951502.
- [7] Feathers M (2007) The deep synergy between testability and good design. https://web.archive.org/web/20071012000838/http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html.

Test code quality

You probably noticed that the amount of JUnit automated test cases that we have written so far is quite significant. This is what happens in practice: the test code base grows up very fast. And like with the production code, **we have to put some extra effort in making high-quality test code bases so that we can maintain and evolve them in a sustainable way**.

In this chapter, we go over some good practices for writing test code. We'll discuss the so-called **test code smells**. These test smells are structures or pieces of code you see in the test code base that indicate problems in the test code or in the system. This is a clear analogy to **code smells**, however focused on test code.

To understand how to write good test code, we first should understand the structure of a test method. Automated tests are very similar in structure. They almost always follow the AAA ("triple A") structure. This acronym stands for **Arrange**, **Act**, and **Assert**. In the **Arrange** phase, we get everything we need to execute the test. This usually means initializing some object and/or setting up some values. Some people call it the "set up of the test". In practice, this is where we write the inputs we devised using any of the testing techniques we discussed. The **Act** phase is where the production code under test is exercised. It is usually done by means of one or many method calls. The result is then used in the **Assert** phase, where we assert that it is what we expect it to be.

We have an automated test:

```
@Test
public void nonLeapCenturialYears() {
    // this is the arrange
1.  int year = 1500;

    // this is the act
2.  LeapYear ly = new LeapYear();
3.  boolean leap = ly.isLeapYear(year);

    // this is the assert
4.  assertFalse(leap);
}
```

This test is made for the `LeapYear` class and tests the `isLeapYear` method. We find the *arrange* part at the start: we define `1500` as the year we want to pass to the production method. Then we use this value as an input to the method under test in lines 2 and 3. This is the *act* part; we use the production code to find a result for a certain input. In line 4, we *assert* that the result is false.

Watch our video on Youtube: https://www.youtube.com/embed/tH_-ilwbDmc

The FIRST principles

Now, let's discuss some best practices in test code. We will discuss the "FIRST Properties of Good Tests". They come originally from the Pragmatic Unit Testing book (see references). Each letter in the acronym represents one best practice.

- **Fast**: In practice, it is really important to run the test after each change, to check that the changes do not break anything. For that to happen smmothly, **the test execution should be fast**. Although we do not have a hard line for when a test is slow or fast, the idea is, however, that when it takes a long time to run a test, developers will not run the test all the time; it would take just too much of their time. In order to keep our tests fast, we should use as few slow dependencies as possible. We have already seen a way to do this with mock objects.
- **Isolated**: When writing a test, we want this specific **test to be as isolated as possible** in a two different perspectives. The first one is in terms of what it tests. The test should check an isolated piece of functionality. Following this, **good unit tests only focus on small pieces of code or in single functionalities**. Another perspective is the isolation from other tests. **Tests should not depend on each other.** When, for example, test A only works when test B runs before, tests are not independent anymore. To avoid such dependency, you have to be careful with shared resources between the tests, e.g., databases, files. The resources have to be the same at the end of a test as they were at the beginning (i.e., your tests should "clean their messes up" at the end of their execution).
- **Repeatable**: **A repeatable test is a test that gives the same result, no matter how many times it is executed.** If a test sometimes fails and sometimes passes, without any changes in the system, you suddenly loose confidence in that test. Repeatable tests are related to isolated tests. A test that is not so well isolated tend to be not repeatable. (We will call these tests, *flaky tests* later on in this chapter).
- **Self-validating/Self-arranging**: **The tests should validate the result themselves.** We write test code to automate the tests, so that we do have to compare the results ourselves. Tests are made self-validating by using assertions. Tests should act as oracles: they should make sure that the program behaved as expeted. When we talk about self-validating, we should also talk about **self-arranging**. This means that the test should reach the required initial state itself. For example, if the production code uses a database and requires that this database is in some state before running, the test should make sure that this database contains the necessary data to run the code.
- **Timely**: The final property is timely. Now, we are not talking just about the code, but also about the development process. Automated tests should be written often and shortly after the production code. If we leave automated tests to the end of the process, e.g., long after the production code is written, chances are that they will not be written at all. We should become "test

infected"! Testing the software should become a habit. (We also discuss the idea of Test-Driven Development, or development guided by the tests in a specific chapter.)

Watch our video on Youtube: https://www.youtube.com/embed/5wLrj-cr9Cs

Test Desiderata

Kent Beck, the "creator" of Test-Driven Development (and author of the fantastic "Test-Driven Development: By Example" book), recently wrote a list of eleven properties that good tests have (the test desiderata).

Directly from his blog post (each link below points to a YouTube video he recorded about the topic):

- Isolated — tests should return the same results regardless of the order in which they are run.
- Composable — if tests are isolated, then I can run 1 or 10 or 100 or 1,000,000 and get the same results.
- Fast — tests should run quickly.
- Inspiring — passing the tests should inspire confidence
- Writable — tests should be cheap to write relative to the cost of the code being tested.
- Readable — tests should be comprehensible for reader, invoking the motivation for writing this particular test.
- Behavioral — tests should be sensitive to changes in the behavior of the code under test. If the behavior changes, the test result should change.
- Structure-insensitive — tests should not change their result if the structure of the code changes.
- Automated — tests should run without human intervention.
- Specific — if a test fails, the cause of the failure should be obvious.
- Deterministic — if nothing changes, the test result shouldn't change.
- Predictive — if the tests all pass, then the code under test should be suitable for production.

Test Smells

Now that we covered some best practices we can start looking at the other side of the coin, **the test smells**.

In production code we use the term **code smells** for indications or symptoms that indicate a deeper problem in the system. Some very well-known examples are *Long Method*, *Long Class*, or *God Class*. A good number of research papers show us that code smells hinder the comprehensibility and the maintainability of software systems.

Code smells can also occur in the test code. We call them **test smells**. Research also shows that these test smells occur a lot in real life and, unsurprisingly, often negatively impact the system. For example, a paper by Bavota and his co-authors shows that the test smells are widely spread and negatively impact the comprehensibility of the code. Another study by Spadini and his co-authors says that tests, that are affected by test smells, have defects and are changed more often than tests that are not affected.

You will see that most of these smells can be mapped to one of the best practices. Throughout this chapter we will go over some test smells and how to avoid them. There are more test smells you can study, though. A good reference for future studies here is the xUnit Test Patterns book, by Meszaros.

One common test smell is **Code Duplication**. It is not surprising that this is a test smell, as it is also very common in production code. We already noted that the tests are all very similar in structure. With that we can very easily get code duplication. Developers often just copy paste the code instead of coming up with some private method, or more generally, abstractions.

The problem with duplicate code is mostly its effect on the maintainability of code. If there is a need for a change in the duplicated piece of code, you will have to apply the change in all the places where the code was duplicated. In practice, it is very easy to forget one of the places, and you end up with problematic code in your tests. This is similar to the impact of code duplication in production code. Because of its effect on the maintainability of a system, code duplication is considered to be a test smell.

Another very common test smell is called **Assertion Roulette**. Assertion roulette is when a test fails, and it is very hard for the developers to understand the failing assertion, i.e., why does it fail? This is problematic, as the developer cannot see what is going wrong exactly and therefore cannot fix the error. Assertion roulette can happen due to a couple of reasons. The first one is the amount of assertions in the test. A test that contains a lot of assertions might make the life of developer more difficult when one assertion fails. After all, understanding the full context of a complex sequence of assertions is just harder than understanding the full context of a simple sequence of assertions. Sometimes just adding a comment, or JUnit message in the assertion can help.

Another widely used strategy is called the "one assertion per method" strategy. With that strategy, we aim to have the least amount of assertions in a test method as possible. This can be taken to the extreme by just allowing exactly one assertion per method. More pragmatically though, when you see a test method with a lot of assertions, you can probably think about splitting it up in a couple separate tests.

The next test smell corresponds to the Fast of FIRST principles: **Slow tests**. As we discussed, test cases should be fast, because then the developers can run the test after each small change they make. Slower tests will be executed less frequently and therefore give less feedback to the developer. When you encounter a slow test you should really try to speed it up. If you have a very slow dependency, you could for example try to mock it.

Another test smell relates to the Self-arranging of FIRST: **Resource Optimism**. Resource optimism happens when a test assumes that a necessary resource (e.g., a database) is readily available at the start of its execution. Again, with the database example: to avoid resource optimism, a test should not assume that the database is already in the correct state, and should set up the state there itself, by means of, for example, INSERT instructions at the beginning of the test method.

Another type of resource optimism is assuming that the resource is available all the time. When we are using a webservice, for example, this might not always be the case, as the webservice might be down for reasons we do not control. To avoid this test smell, you have two options: One is again to avoid using external resources, by using mock objects. If you cannot avoid using the external dependencies, you can make the test robust enough. In that case, you can skip the test when the resource is unavailable, and provide a message explaining why the test was skipped. This is, at least, better than letting the test fail for a bad reason.

In addition to changing your tests, you have to make sure that all the environments that the tests are executed have the required resources available. Continuous integration tools like Jenkins, CircleCI, and Travis can help you in making sure that you run the tests always in the same environment.

The next test smell is **Test Run War**. This happens when the tests pass if you execute them alone, but fail as soon as someone else runs the test at the same time. This often happen when the tests consume the same resource, e.g., the same database. Imagine a test suite that communicates to a global database. When developer 1 runs the test, the test changes the state of the database; at the same time, developer 2 runs the same test that also goes to the same database. Now, both tests are touching the same database at the same time. This unexpected situation which might make the test to fail. If you feel like we talked about this before, you are correct; this smell is highly related to the *Isolation* of the FIRST princinples. When you encounter a Test Run War, make sure to isolate them well.

Another test smell that we often encounter is the **General Fixture**. A fixture basically means the arrange part of the test, where we create all the objects and inputs needed by the test. We often see developers creating one large method (using, for example, the `@BeforeEach` annotation of JUnit) that creates *all* the objects needed by the *all* tests. Having such a fixture is not necessarily a problem. However, when test A uses just part of this fixture, and part B uses just another part, this means that this fixture is too generic. The problem with a generic fixture is that it becomes very hard to understand what a single test really needs and uses. To resolve this issue, you have to make sure that the tests contain clear and concise fixures, nothing more than what it really needs. Explore the design pattern called **Test Data Builder**, that helps us in avoiding this test smell.

The **Indirect tests** smell concerns the class that is tested by a test class. When the test goes beyond testing the class under test, but also tests other classes, we call it an *indirect test*. This might problematic due to the change propagation that might happen in this test. Imagine a test class `ATest` that tests not only class `A`, but also class `B`. If a bug exists in class `B`, we expect tests in `BTest` to fail; however, the tests for class `A` in `ATest` will also break, due to this indirect testing.

These extra failing tests are cumbersome and might cost a good amount of time from the developer. We should try as much as possible to keep our tests focused, making sure that they do not indirectly test other classes.

Finally, we have the test smell called **Sensitive Equality**. We use assertions to verify that the production code behaves as expected. This test smell is when we have assertions that are too strict. Or, in other words, too sensitive. We want our tests to be as resilient as possible, and only break if there is indeed a bug in the system. We do not want small changes that do not affect the behavior of the class to break our tests.

A clear example of such a smell is when the developer decides to use the results of a `toString()` method to assert the expected behavior. Imagine an assertion like `Assertion.assertTrue(invoice.toString().contains("Playstation"))`. `toString()`s change quite often; but a change in `toString()` should not break all the tests... If the developer had made the assertion to focus on the specific property, e.g., `Assertion.assertEquals("Playstation", invoice.getProduct().getName())`, this would not had happened.

Again, all these test smells are covered more in-depth in XUnit Test Patterns by Gerard Meszaros. We highly recommend you to read that book!

Watch our video on Youtube: https://www.youtube.com/embed/QE-L818PDjA

Watch our video on Youtube: https://www.youtube.com/embed/DLfeGM84bzg

Flaky Tests

It is unfortunately quite common to have tests in our test suite that sometimes passes, but sometimes fails. We do not like these tests; after all, if a test gives us false positives, we can not really trust on them. We call such tests, **flaky tests**.

Flaky tests can have a lot of causes. We will name a few examples:

- A test could be flaky, because it **depends on an external and/or shared resource**. Let's say we need a database to run our tests. Sometimes the test passes, because the database is available; sometimes it fails, because then the database is not available. In this case, the test depends on the availability of an external infrastructure.

- The tests can also be flaky because of time-outs. This cause is common in web applications. With timeouts, the test waits for something to happen in the web application, e.g., a new message to appear in an HTML element. If the web application is a bit slower than normal, the test will suddenly fail.
- Finally we could have interacting tests, which we also discussed above. Then test A influences the result of test B, possibly causing it to fail. All these examples are plausible causes of flakiness that actually occur in practice.

As you might have notices, a lot of the causes of flaky tests correspond to scenarios we described when discussing the test smells. The quality of our test code is indeed very important!

If you want to find the exact cause of a flaky test, the author of the XUnit Test Patterns book has made a whole decision table. You can find it in the book or on Gerard Meszaros' website here. With the decision table you can find a probable cause for the flakiness of your test. For example: Does the flaky test get worse over time? This means if the test gets slower and slower to execute. If this is true, you probably have a resource leakage in the tests, otherwise it is probably a non-deterministic test.

If you want to read more about flaky tests, we suggest the following papers and blog posts (including Google discussing how problematic flaky tests are for their development teams):

- Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014, November). An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 643-653). ACM. Authors' version: http://mir.cs.illinois.edu/~eloussi2/publications/fse14.pdf
- Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018, May). D e F laker: automatically detecting flaky tests. In Proceedings of the 40th International Conference on Software Engineering (pp. 433-444). ACM. Authors' version: http://mir.cs.illinois.edu/legunsen/pubs/BellETAL18DeFlaker.pdf
- Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019, April). iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST) (pp. 312-322). IEEE. Authors' version: http://taoxie.cs.illinois.edu/publications/icst19-idflakies.pdf
- Listfield, J. Where do our flaky tests come from? Link: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, 2017.
- Micco, J. Flaky tests at Google and How We Mitigate Them. Link: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, 2017.
- Fowler, M. Eradicating Non-Determinism in Tests. Link: https://martinfowler.com/articles/nonDeterminism.html, 2011.

Watch our video on Youtube: https://www.youtube.com/embed/-OQgBMSBL5c

Improving the Readability of Test Code

As developers are often working on their test code bases. Therefore, it is crucial that the developers can understand the test code easily. **We need readable and understandable test code.** Note that "readability" is one of the test desiderata! But how can we do this? In the following, we give you some tips to write readable test code.

- The first tip concerns the **structure of your tests**. As you have seen earlier, tests all follow the same structure: Arrange, Act and Assert. When **these are clearly separated, it is easier for a developer to see what is happening in the test**. In the example, at the start of this chapter, we did this by just adding some blank lines between the different parts.
- A second tip is about the **information in tests**. Test are full of information, i.e., input values, expected results. However, we often have to deal with complex data structures and information, which makes the test code naturally complex. **We should make sure that the important information present in a test is easy to understand.** An easy way to do this is by storing the values in variables with descriptive names. This is illustrated in the example below.

We have written a test for an invoice that checks if the invoice has been paid.

```
@Test
public void invoiceNotPaid() {
  Invoice inv = new Invoice(1, 10.0, 25.0);

  boolean p = inv.isPaid();

  assertFalse(p);
}
```

As you can see, is not clear what all the hard-coded values really mean (what does 1, 10.0, or 25.0 mean?), and thus, it is not clear what kind of information this test deals with.

Let's rewrite the test with some variables.

```
@Test
public void invoiceNotPaid() {
  int anyId = 1;
  double tax = 10.0;
  double amount = 25.0;
  Invoice inv = new Invoice(anyId, tax, amount);

  boolean paid = inv.isPaid();

  assertFalse(paid);
}
```

The variables with descriptive names makes it easier for a developer to understand what they actually mean. Now, we clearly see that the ID can be any number (does not really matter which one for this test), hence the `any` at the front. The variables `tax` and `amount` also now clearly explain what the previously "magic numbers with no explanation" mean.

- Finally we have a small tip for assertions. When an assertions fails, a developer should be able to quickly see what is going wrong in order to fix the fault in the code. If the assertion in your test is complex, you might want to express it in a different way. For example, you might create a helper that performs this assertion step by step, so that a developer can easily follow the idea. Or you might want to add an explanatory message to the assertion. Or, even, why not a code comment? Frameworks like AssertJ have become really popular among developers, due to its expresiveness in assertions.

TODO: Example here of assertj

Watch our video on Youtube: https://www.youtube.com/embed/RlqLCUl2b0g

Exercises

Exercise 1. Jeanette just heard that two tests are behaving strangely: when executed in isolation, both of them pass. However, when executed together, they fail. Which one of the following **is not** cause for this?

1. Both tests are very slow.
2. They depend upon the same external resources.
3. The execution order of the tests matter.
4. They do not perform a clean-up operation after execution.

Exercise 2. RepoDriller is a project that extracts information from Git repositories. Its integration tests consumes lots of real Git repositories, each one with a different characteristic, e.g., one repository contains a merge commit, another repository contains a revert operation, etc.

Its tests look like what you see below:

```java
@Test
public void test01() {

  // arrange: specific repo
  String path = "test-repos/git-4";

  // act
  TestVisitor visitor = new TestVisitor();
  new RepositoryMining()
  .in(GitRepository.singleProject(path))
  .through(Commits.all())
  .process(visitor)
  .mine();

  // assert
  Assert.assertEquals(3, visitor.getVisitedHashes().size());
  Assert.assertTrue(visitor.getVisitedHashes().get(2).equals("b8c2"));
  Assert.assertTrue(visitor.getVisitedHashes().get(1).equals("375d"));
  Assert.assertTrue(visitor.getVisitedHashes().get(0).equals("a1b6"));
}
```

Which test smell does this piece of code suffers from?

1. Mystery guest
2. Condition logic in test
3. General fixture
4. Flaky test

Exercise 3. In the code below, we present the source code of an automated test. However, Joe, our new test specialist, believes this test is smelly and it can be better written. Which of the following could be Joe's main concern?

1. The test contains code that may or may not be executed, making the test less readable.
2. It is hard to tell which of several assertions within the same test method will cause a test failure.
3. The test depends on external resources and has nondeterministic results depending on when/where it is run.
4. The test reader is not able to see the cause and effect between fixture and verification logic because part of it is done outside the test method.

```
@Test
public void flightMileage() {
  // setup fixture
  // exercise contructor
  Flight newFlight = new Flight(validFlightNumber);
  // verify constructed object
  assertEquals(validFlightNumber, newFlight.number);
  assertEquals("", newFlight.airlineCode);
  assertNull(newFlight.airline);
  // setup mileage
  newFlight.setMileage(1122);
  // exercise mileage translater
  int actualKilometres = newFlight.getMileageAsKm();
  // verify results
  int expectedKilometres = 1810;
  assertEquals(expectedKilometres, actualKilometres);
  // now try it with a canceled flight
  newFlight.cancel();
  boolean flightCanceledStatus = newFlight.isCancelled();
  assertFalse(flightCanceledStatus);
}
```

Exercise 4. See the test code below. What is the most likely test code smell that this piece of code presents?

```
@Test
void test1() {
  // webservice that communicates with the bank
  BankWebService bank = new BankWebService();

  User user = new User("d.bergkamp", "nl123");
  bank.authenticate(user);
  Thread.sleep(5000); // sleep for 5 seconds

  double balance = bank.getBalance();
  Thread.sleep(2000);

  Payment bill = new Payment();
  bill.setOrigin(user);
  bill.setValue(150.0);
  bill.setDescription("Energy bill");
  bill.setCode("YHG45LT");

  bank.pay(bill);
  Thread.sleep(5000);

  double newBalance = bank.getBalance();
  Thread.sleep(2000);

  // new balance should be previous balance - 150
  Assertions.assertEquals(newBalance, balance - 150);
}
```

1. Flaky test.
2. Test code duplication.
3. Obscure test.
4. Long method.

Exercise 5. In the code below, we show an actual test from Apache Commons Lang, a very popular open source Java library. This test focuses on the static `random()` method, which is responsible for generating random characters. A very interesting detail in this test is the comment: *Will fail randomly about 1 in 1000 times*.

```
/**
 * Test homogeneity of random strings generated --
 * i.e., test that characters show up with expected frequencies
 * in generated strings.  Will fail randomly about 1 in 1000 times.
 * Repeated failures indicate a problem.
 */
@Test
public void testRandomStringUtilsHomog() {
    final String set = "abc";
    final char[] chars = set.toCharArray();
    String gen = "";
    final int[] counts = {0,0,0};
    final int[] expected = {200,200,200};
    for (int i = 0; i< 100; i++) {
       gen = RandomStringUtils.random(6,chars);
       for (int j = 0; j < 6; j++) {
           switch (gen.charAt(j)) {
               case 'a': {counts[0]++; break;}
               case 'b': {counts[1]++; break;}
               case 'c': {counts[2]++; break;}
               default: {fail("generated character not in set");}
           }
       }
    }
    // Perform chi-square test with df = 3-1 = 2, testing at .001 level
    assertTrue("test homogeneity -- will fail about 1 in 1000 times",
        chiSquare(expected,counts) < 13.82);
}
```

Which one of the following **is incorrect** about the test?

1. The test is flaky because of the randomness that exists in generating characters.
2. The test checks for invalidly generated characters, and that characters are picked in the same proportion.
3. The method being static has nothing to do with its flakiness.
4. To avoid the flakiness, a developer could have mocked the random function.

References

- Chapter 5 of Pragmatic Unit Testing in Java 8 with Junit. Langr, Hunt, and Thomas. Pragmatic Programmers, 2015.
- Meszaros, G. (2007). xUnit test patterns: Refactoring test code. Pearson Education.
- Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., & Binkley, D. (2012, September). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In 2012 28th IEEE International Conference on Software Maintenance (ICSM) (pp. 56-65). IEEE.

- Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., & Binkley, D. (2015). Are test smells really harmful? An empirical study. Empirical Software Engineering, 20(4), 1052-1094.
- Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014, November). An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 643-653). ACM.
- Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018, May). D e F laker: automatically detecting flaky tests. In Proceedings of the 40th International Conference on Software Engineering (pp. 433-444). ACM.
- Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019, April). iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST) (pp. 312-322). IEEE.
- Listfield, J. Where do our flaky tests come from?
  Link: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, 2017.
- Micco, J. Flaky tests at Google and How We Mitigate Them.
  Link: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, 2017.
- Fowler, M. Eradicating Non-Determinism in Tests. Link: https://martinfowler.com/articles/nonDeterminism.html, 2011.

Mutation testing

How do we know if we tested enough? For example, in the structural-based testing chapter, we discussed line coverage, branch coverage, and MC/DC. In the model-based testing chapter, we discussing transition coverage and path coverage. All these **adequacy criteria** measure how much of the program is exercised by the tests we devised.

However, these criteria alone might not enough to determine the quality of the test cases. In practice, we can exercise a large part of the system, while testing very little.

Suppose a simple class with a single method:

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0) {
      return null;
    }
    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

Now, imagine a tester writing the following test cases:

```
@Test
public void testGetValues() {
  int[] values = Division.getValues(1, 1);
}

@Test
public void testZero() {
  int[] values = Division.getValues(1, 0);
}
```

These tests gets us to 100% branch coverage. However, you probably also noticed that something is missing in these tests: **the assertions**! These tests will never fail!

Fault Detection Capability

Let's discuss one more adequacy criterion: the **fault detection capability**. It indicates the test's capability to reveal faults in the system under test. The more faults a test can detect, in other words, the more faults a test fails on, the higher its fault detection capability. Using this criterion, we can indicate the quality of our test suite in a better way than with just the coverage metrics we have so far.

The fault detection capability does not just regard the amount of production code executed, but also the assertions made in the test cases. For a test to be adequate according to this criterion, it has to have a meaningful **test oracle** (i.e., meaningful assertions).

The fault detection capability, as a test adequacy criterion, is the fundamental idea behind **mutation testing**. In mutation testing, we change small parts of the code, and check if the tests can find the introduced fault.

In the previous example, we made a test suite that was not adequate at all, according to the fault detection capability. As there were no assertions, the tests would never find any faults in the code.

```
@Test
public void testGetValues() {
  int[] values = Division.getValues(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}

@Test
public void testZero() {
  int[] values = Division.getValues(1, 0);
  assertNull(values);
}
```

Tests with assertions check if the result of the method is what we expect. We have a **test oracle** now.

To see how the values in a test case influence the fault detection capability, let's create two tests, where the denominator is not 0.

```
@Test
public void testGetValuesOnes() {
  int[] values = Division.getValues(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}

@Test
public void testGetValuesDifferent() {
  int[] values = Division.getValues(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
```

For the fault detection capability, we want to see if the tests catch any faults that could be in the code.

This means we have to go back to the source code and introduce an error:

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0) {
      return null;
    }
    int quotient = a * b; // the bug was introduced here
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

We replace the division by a multiplication; a clear bug.

If we run our tests with the buggy code, we see that the test with the values 1 and 1 (the `testGetValuesOnes()` test) still passes, but the other test (the `testGetValuesDifferent()`) fails. This indicates that the second test has a higher fault detection capability.

Even though both tests exercise the method in the same way and execute the same lines, we see a difference in the fault detection capability. This is because of the different input values for the method and the different test oracles (assertions) in the tests. In this case, the input values and test oracle of the `testGetValuesDifferent()` test can better detect that bug.

Watch our video on Youtube: https://www.youtube.com/embed/QYbqz-gFWAk

Hypotheses for Mutation Testing

We are now ready to generalize the idea. The idea of mutation testing is to **assess the quality of the test suite**. This is done by manipulating the source code a bit and running the tests with this manipulated source code. If we have a good test suite, at least one of the tests will fail on this changed (buggy) code. Following this procedure, we get a sense of the fault error capability of our test suite.

In mutation testing we use **mutants**. The mutants are the defects, or faults, that we introduce in the source code and then use to determine the quality of the test suite.

A big question regarding the mutants is what their size should be. We can change single operations, whole lines or even multiple lines of code. What would work best?

Mutation testing and the answer to this question are based on the following two hypotheses:

- **The Competent Programmar Hypothesis (CPH)**: Here, we assume that the program is written by a competent programmer. More importantly, this means that given a certain specification, the programmer creates a program that is either correct, or it differs from a correct program by a combination of simple errors.

- **The Coupling Effect**: The coupling effect hypothesis states that simple faults are coupled to more complex faults. In other words, test cases that detect simple faults, will also detect complex faults.

Based on these two hypotheses, we can determine the size that the mutants should have. Realistically, following the competent programmer hypothesis, the faults in actual code will be small. This indicates that the mutants' size should be small as well. Considering the coupling effect, test cases that detect small errors, will also be able to detect larger, more complex errors.

Terminology

To talk about mutation testing in a depth way, let's define some terms:

- **Mutant**: Given a program P, a mutant called P' is obtained by introducing a *syntactic change* to P. A mutant is killed if a test fails when executed with the mutant.
- **Syntactic Change**: A small *change* in the code. Such a small change should make the code still valid, i.e., the code can still compile and run.
- **Change**: A change, or alternation, to the code that mimic typical human mistakes. We will see some examples of these mistakes later.

We illustrate mutation testing with these concepts in the example below.

Suppose we have a `Fraction` class with a method `invert()`.

```
public class Fraction {
  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1.  if (numerator == 0) {
2.    throw new ArithmeticException("...");
    }
3.  if (numerator == Integer.MIN_VALUE) {
4.    throw new ArithmeticException("...");
    }
5.  if (numerator < 0) {
6.    return new Fraction(-denominator, -numerator);
    }
7.  return new Fraction(denominator, numerator);
  }
}
```

We have a small test suite for this method as well.

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.00001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.00001);
}

@Test
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  assertThrows(ArithmeticException.class, () -> f.invert());
}

@Test
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  assertThrows(ArithmeticException.class, () -> f.invert());
}
```

We have two tests for some corner cases that throw an exception, and two more "happy path" tests. Now, we want to determine the quality of our test suite, using mutation testing.

First, we have to create a *mutant* by applying a *syntactic change* to the original method. Keep in mind that, because of the two hypotheses, we want the syntactic change to be small: one operation/variable should be enough. Moreover, the syntactic change is a *change*, hence it should mimic mistakes that could be made by a programmer.

For the first mutant we change line 6. Instead of `-numerator` we just say `numerator`.

The mutant looks like as follows:

```
public class Fraction {
  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1.  if (numerator == 0) {
2.    throw new ArithmeticException("...");
    }
3.  if (numerator == Integer.MIN_VALUE) {
4.    throw new ArithmeticException("...");
    }
5.  if (numerator < 0) {
6.    return new Fraction(-denominator, numerator);
    }
7.  return new Fraction(denominator, numerator);
  }
}
```

If we would execute the test suite on this mutant, the `testInvertNegative()` test will fail, as `result.getFloat()` would be positive instead of negative.

Another mistake could be made in line 1. When we studied boundary analysis, we saw that it is important to test the boundaries due to off-by-one errors. We can make a syntactic change by introducing such an off-by-on error. Instead of `numerator == 0`, in our new mutant we make it `numerator == 1`:

```
public class Fraction {
  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1.  if (numerator == 1) {
2.    throw new ArithmeticException("...");
    }
3.  if (numerator == Integer.MIN_VALUE) {
4.    throw new ArithmeticException("...");
    }
5.  if (numerator < 0) {
6.    return new Fraction(-denominator, numerator);
    }
7.  return new Fraction(denominator, numerator);
  }
}
```

We see that, again, the test suite catches this error. The test `testInvertZero()` will fail, as it expects an exception, but none is thrown in the mutant.

Automation

Manually writing the mutations of our programs takes a lot of time, and we probably would only think of the cases that are already tested. Like with test execution, we want to **automate the mutation process**. There are various tools that automatically generate mutants for mutant testing, but they all use the same methodology.

First we need mutation operators. A **mutation operator** is a grammatic rule that can be used to introduce a syntactic change. This means that, if the generator sees a statement in the code that corresponds to the grammatic rule of the operator (e.g., `a + b`), then the mutation operator specifies how to change this statement with a syntactic change (e.g., turning it into `a - b` for example).

We distinguish two categories of mutation operators:

- **Real fault based operators**: Operators that are very similar to defects seen in the past for the same kind of code. Such operators look like common mistakes made by programmers in similar code.
- **Language-specific operators**: Mutations that are made specifically for a certain programming language. For example, changes related to the inheritance feature we have in the Java language, or changes regarding pointer manipulations in C, which we cannot simply apply to all the languages.

Most mutation testing tools include various basic mutation operators for real fault based operators. We briefly go over some common mutation operators:

- **AOR - Arithmetic Operator Replacement**: Replaces an arithmetic operator by another arithmetic operator. Arithmetic operators are `+`, `-`, `*`, `/`, `%`.
- **ROR - Relational Operator Replacement**: Replaces a relational operator by another relational operator. Relational operators are `<=`, `>=`, `!=`, `==`, `>`, `<`.
- **COR - Conditional Operator Replacement**: Replaces a conditional operator by another conditional operator. Conditional operators are `&&`, `||`, `&`, `|`, `!`, `^`.
- **AOR - Assignment Operator Replacement**: Replaces an assignment operator by another assignment operator. Assignment operators include `=`, `+=`, `-=`, `/=`.
- **SVR - Scalar Variable Replacement**: Replaces each variable reference by another variable reference that has been declared in the code.

For each of the mutation operators, we provide an example. We first show the original code, and then the mutant that could be given by the mutant operator.

Arithmetic Operator Replacement

Original:

```
int c = a + b;
```

Example of a mutant:

```
int c = a - b;
```

Relational Operator Replacement

Original:

```
if (c == 0) {
  return -1;
}
```

Example of a mutant:

```
if (c > 0) {
  return -1;
}
```

Conditional Operator Replacement

Original:

```
if (a == null || a.length == 0) {
  return new int[0];
}
```

Example of a mutant:

```
if (a == null | a.length == 0) {
  return new int[0];
}
```

Assignment Operator Replacement

Original:

```
c = a + b;
```

Example of a mutant:

```
c -= a + b;
```

Scalar Variable Replacement

Original:

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0 || b == Integer.MIN_VALUE){
      return null;
    }

    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

Example of a mutant:

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (a == 0 || a == Integer.MIN_VALUE){
      return null;
    }

    int quotient = b / a;
    int remainder = quotient % a;

    return new int[] {remainder, a};
  }
}
```

Specifically to Java, there are a lot of language-specific operators. We can, for example, change the inheritance of the class, remove an overriding method, or change some declaration types. We will not go into detail about these language-specific mutant operators, but some examples are:

- Access Modifier Change
- Hiding Variable Deletion
- Hiding Variable Insertion
- Overriding Method Deletion
- Parent Constructor Deletion
- Declaration Type Change

Of course, there exist many more mutant operators that are used by mutant generators. For now, you should at least have an idea what mutation operators are, how they work and what we can use them for.

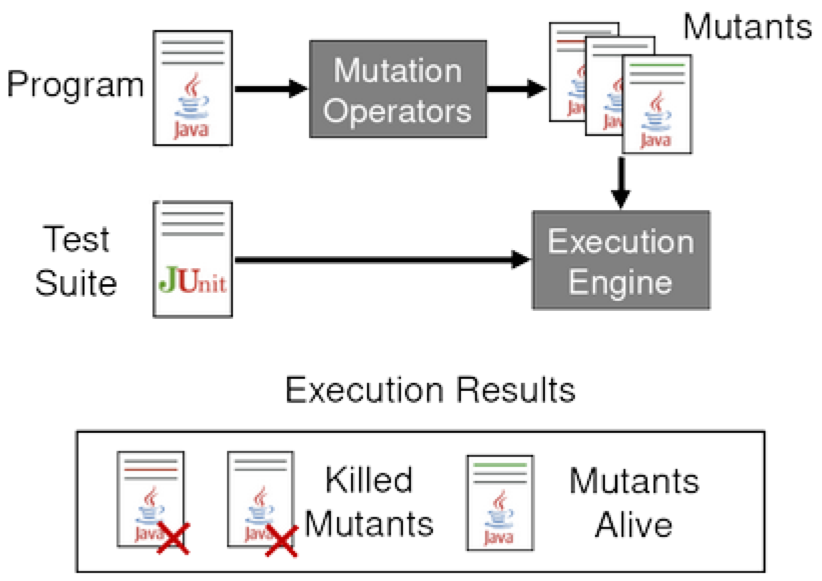
Watch our video on Youtube:
https://www.youtube.com/embed/KXQTWLyR5CA

Mutation Analysis and Testing

Now that the concept of mutation testing and how to make it is clear, let's look at how we can use mutation testing in practice.

Our goal is to use mutation testing to determine the quality of our test suite. We have seen that we first have to create the mutants and that it is best to do this in an automated way with the help of mutant operators. Then, we run the test suite against the created mutants with an execution engine. If one of the tests fails, we say that the test suite kills the mutant. This is good, as it suggests that our test suite has some fault detection capability. If none of the tests in our test suite fails when executed against the mutant, this mutant stays alive.

This process is illustrated in the diagram below:



When performing mutation testing, we count the number of mutants our test suite killed and the number of mutants that were still alive. By counting the amount of each of these mutant groups, we can give a value to the quality of our test suite.\
We define the **Mutation Score** as:

$$\text{Mutation score} = \frac{\text{killed mutants}}{\text{mutants}}$$

Computing this mutation score is what we call **mutation analysis**. More formally:\
Mutation analysis means assessing the quality of a test suite, by computing its mutation score.

When the mutation score is low, we might want to change or add new test cases in our test suite. This is called **mutation testing**. Again, the definition: **Mutation testing** means improving the quality of the test suite using mutants (by adding and/or changing test cases).

These concepts are very related to each other. To do mutation testing, you have to compute the mutation score first. This then indicates whether the test suite should be changed. If the mutation score is low, there are a lot of mutants that are not killed by the test suite. Then, it is indeed our job to improve the test suite.

Equivalent Mutants

Calculating the mutation score is, in practice, challenging. Why? The mutation score increases when less mutants are alive. This suggests that the best scenario is to have all the mutants killed by the test suite. While this is indeed the best scenario, it is often unrealistic. Some of the mutants are impossible to kill and will then always stay alive.

The mutants that cannot be killed are called equivalent mutants. An **equivalent mutant** is a mutant that always behaves as the original program. If the mutant behaves like the normal code, it will always give the same output as the original program for any given input. Of course, this makes this mutant (which is basically the same program as the one under test) impossible to be killed by the tests.

Here, the equivalence is related to the definition of program equivalence. Program equivalence roughly means that two programs are functionally equivalent when they produce the same output for every possible input. This is also the equivalence between the normal code and an equivalent mutant.

Let's have a look at the following method. We left some irrelevant parts out.

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index == 0)
      break;
  }
}
```

Our mutation testing tool generates a mutant using relational operator replacement. The mutant is as follows:

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index <= 0)
      break;
  }
}
```

Note how the original code decremented `index`, which started at 10, and breaked from the loop when index would be equal to 0. The mutant works exactly the same, even though the condition is technically different. `index` is still decremented from 10 to 0. Because `index` will never be negative, the `==` operator does the same as the `<=` operator.

The mutant produced by the generator is an equivalent mutant in this case.

Because of these equivalent mutants, we need to change the mutation score formula. We do not want to take the equivalent mutants into account, as there is nothing wrong with the tests when they do not kill these mutants.

The new formula becomes:

$$\text{Mutation score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}}$$

For the denominator, we just count the amount of non-equivalent mutants, instead of all the mutants. To compute this new mutation score automatically, we would need a way to automatically determine whether a mutant is an equivalent mutant. Unfortunately, we cannot do this automatically. **Detecting equivalent mutations is an undecidable problem**. We can never be sure that a mutant behaves the same as the original program for every possible input.

Application

Mutation testing sounds like a great way to analyse and improve our test suites. The question is, however, if we can actually use mutation testing in practice. For example, we can ask ourselves whether a test suite with a higher mutation score actually finds more errors.

A lot of research done in software engineering tries to bring some insights to this problem. All the existing studies about mutation testing showed that mutants can indeed give a good indication for a test suite's fault detection capability, as long as the mutant operators are carefully selected and the equivalent mutants are removed.

More specifically, a study by Just et al. shows that mutant detection is positively correlated with real fault detection. In other words, the more mutants a test suite detects, the more real faults the test suite can detect as well. Even more interesting is that this correlation is independent from the coverage. Furthermore, the correlation between mutant detection and fault detection is higher than the correlation between statement coverage and fault detection. So, the mutation score provides a better measure for the fault detection capability than the test coverage.

Cost

Of course, mutation testing is not without its costs. We have to generate the mutants, possibly remove the equivalent mutants, and execute the tests with each mutant. In fact, mutation testing is quite expensive, i.e., it takes a long time to perform.

Let's assume we want to do some mutation testing. We have:

- A code base with 300 Java classes
- 10 test cases for each class
- Each test case takes 0.2 seconds on average
- The total test execution time is then: $300 \cdot 10 \cdot 0.2 = 600s (10 \text{ min})$

This execution time is for just the normal code. For the mutation testing, we decide to generate on average 20 mutants per class.

We will have to execute the entire test suite of a class on each of the mutants. Per class, we need $20 \cdot 10 \cdot 0.2 = 40$ seconds. In total, the mutation testing will take $300 \cdot 40 = 12000$ seconds, or 3 hours and 20 minutes.

Indeed mutation testing can take a very long time.

Because of this cost, researchers have tried to find ways to make mutation testing faster for a long time. Based on some observations, they came up with a couple of heuristics.

- The first observation is that a test case can never kill a mutant if it does not cover the statement that changed (also known as the *reachability condition*). Based on this observation, we only have to run the test cases that cover the changed statement. This reduces the amount of test cases to run and, with that, the execution time. Furthermore, once a test case kills a mutant, we do not have to run the other test cases anymore. This is because the test suite needs at least one test case that kills the mutant. The exact number of test cases killing the mutant does not really matter.

- A second oberservation is that mutants generated by the same operator and injected at the same location are likely to be coupled to the same type of fault. This means that when we use a certain mutation operator (Arithmetic Operator Replacement, for example) and we replace the same statement with this mutant operator, we get two mutants that represent the same fault in the code. It is then highly likely that if the test suite kills one of the mutants, it will also kill the others. An heuristic that follows from this observation is to run the test suite against a subset of all the less mutants (a technique also known as *do fewer*). Obviously, when we run the test suite against a smaller number of mutations, the overall testing will take less time. The simplest way of selecting the subset of mutants is by means of random sampling. As the name suggests, we just pick random mutants to consider. This is a very simple, yet effective way to reduce the execution time.

There exist other heuristics to decrease the execution time, like e-selective or cluster mutants and operators, but we will not go into detail of those heuristics.

Tools

To perform mutation testing you can use one of many publicly available tools. These tools are often made for specific programming languages. One of the most mature mutation testing tools for Java is called PIT or pitest.

PIT can be run from the command line, but it is also integrated in most popular IDEs (like Eclipse or IntelliJ). Project management tools like Maven or Gradle can also be configured to run PIT.

As PIT is a mutation testing tool, it generates the mutants and runs the test suites against these mutants. Then it generates easy to read reports based on the results. In these reports, you can see the line coverage and mutation score per class. Finally, you can also see more detailed results in the source code and check which individual mutants were kept alive. Try it out!

Watch our video on Youtube:
https://www.youtube.com/embed/BEBhTtSZAlw

Exercises

Exercise 1. "*Crimes* happen in a *city*. One way for us to know that the *police* is actually able to detect these *crimes*, we can *simulate crimes* and see whether the *police* is able to detect them."

In the analogy above, we can replace crimes by bugs, city by software, and police by test suite. What should we replace **simulate crimes** by?

1. Mutation testing
2. Fuzzing testing
3. Search-based software testing
4. Combinatorial testing

TODO: We need to develop more exercises for this chapter

References

- Chapter 16 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., & Fraser, G. (2014, November). Are mutants a valid substitute for real faults in software testing?. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 654-665). ACM.

Fuzzing testing

There is no need for us to develop material about it. Just read the https://www.fuzzingbook.org!

Answers to the exercises

Principles of software testing

Exercise 1

1. Failure, the user notices the system/program behaving incorrectly.
2. Fault, this is a problem in the code, that is causing a failure in this case.
3. Error, the human mistake that created the fault.

Exercise 2

The absence-or-error fallacy. While the software does not have a lot of bugs, it is not giving the user what they want. In this case the verification was good, but they need work on the validation.

Exercise 3

Exhaustive testing is impossible.

Exercise 4

Test early, although an important principle, is definitely not related to the problem of only doing unit tests. All others help people in understanding that variation, different types of testing, is important.

Introduction to software testing automation

Writing tests is fun, isn't it?

Specification-based testing

Exercise 1

A group of inputs that all make a method behave the same way.

Exercise 2

We use the concept of equivalence partitioning to determine which tests can be removed. According to equivalence partitioning we only need to test one test case in a certain partition.

We can group the tests cases in their partitions:

- Divisible by 3 and 5: T1, T2
- Divisible by just 3 (not by 5): T4
- Divisible by just 5 (not by 3): T5
- Not divisible by 3 or 5: T3

Only the partition where the number is divisible by both 3 and 5 has two tests. Therefore we can only remove T1 or T2.

Exercise 3

Following the category partition method:

1. Two parameters: key and value
2. The execution of the program does not depend on the value; it always inserts it into the map. We can define different characteristics of the key:
    - The key can already be present in the map or not.
    - The key can be null.
3. The requirements did not give a lot of parameters and/or characteristics, so we do not have to add constraints.
4. The combinations are each of the possibilities for the key with any value, as the programs execution does not depend on the value. We end up with three partitions:
    - New key
    - Existing key
    - null key

Exercise 4

We go over each given partition and identify whether it is a valid partition:

1. Valid partition: Invalid numbers, which are too small.
2. Valid partition: Valid numbers
3. Invalid partition: Contains some valid numbers, but the range is too small to cover the whole partition.
4. Invalid partition: Same reason as number 3.
5. Valid partition: Invalid numbers, that are too large.
6. Invalid partition: Contains both valid and invalid letters (the C is included in the domain).
7. Valid partition: Valid letters.
8. Valid partition: Invalid letters, past the range of valid letters.

We have the following valid partitions: 1, 2, 5, 7, 8.

Exercise 5

- P1: Element not present in the set
- P2: Element already present in the set
- P3: NULL element.

The specification clearly explicits the three different cases of the correct answer.

Exercise 6

Option 4 is the incorrect one. This is a functional based technique. No need for source code.

Exercise 7

Possible actions:

1. We should treat pattern size 'empty' as exceptional, and thus, test it just once.
2. We should constraint the options in the 'occurences in a single line' category to happen only if 'occurences in the file' are either exactly one or more than one. % It does not make sense to have none occurences in a file and one pattern in a line.
3. We should treat 'pattern is improperly quoted' as exceptional, and thus, test it just once.

Boundary testing

Exercise 1

The on-point is the value in the conditions: `half` .

When `i` equals `half` the condition is false. Then the off-point makes the condition true and is as close to `half` as possible. This makes the off-point `half` - 1.

The in-points are all the points that are smaller than half. Practically they will be from 0, as that is what `i` starts with.

The out-points are the values that make the condition false: all values equal to or larger than `half` .

Exercise 2

The decision consists of two conditions, so we can analyse these separately.

For `n % 3 == 0` we have an on point of 3. Here we are dealing with an equality; the value can both go up and down to make the condition false. As such, we have two off-points: 2 and 4.

Similarly to the first condition for `n % 5 == 0` we have an on-point of 5. Now the off-points are 4 and 6.

Exercise 3

The on-point can be read from the condition: 570.

The off-point should make the condition false (the on-point makes it true): 571.

An example of an in-point is 483. Then the condition evaluates to true.

An example of an out-point, where the condition evaluates to false, is 893.

Exercise 4

Boundary conditions for (numberOfPoints <= 570 && numberOfLives > 10) || energyLevel == 5									
			test cases (x, y)						
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6	t7
numberOfPoints	<= 570	on	570						
		off		571					
	typical	in			20	364	95	283	74
numberOfLives	> 10	on			10				
		off				11			
	typical	in	15	94					
		out					4	3	8
energyLevel	5	on					5		
		off						4	
		off							6
	typical	out	3	7	49	36			

Note that we require 7 test cases in total: `numberOfPoints <= 570` and `numberOfLives > 10` each have one on- and one off-point. `energyLevel == 5` is an equality, so we have two off-points and one on-point. This gives a total of 7 test cases.

For one of the first two conditions we need two typical rows. \ Let's rewrite the whole condition to: `(c1 && c2) || c3`.

To test `c1` we have to make `c2` true, otherwise the result will always be false. \ The same goes for testing `c2` and then making `c1` true.

However, when testing `c3`, we need to make `(c1 && c2)` false, otherwise the result will always be true. That is why, when testing `c3`, `c1` or `c2` has to be false, i.e. and out-point instead of an in-point. Therefore we use two different typical rows for the `numberOfLives` variable. The same could have been done with two typical rows for the `numberOfPoints` variable.

Exercise 5

An on-point is the (single) number on the boundary. It may or may not make the condition true. The off point is the closest number to the boundary that makes the condition to be evaluated to the opposite of the on point. Given it's an inequality, there's only a single off-point.

Exercise 6

on point = 1024, off point = 1025, in point = 1028, out point = 512

The on point is the number precisely in the boundary = 1024. off point is the closest number to the boundary and has the opposite result of on point. In this case, 1024 makes the condition false, so the off point should make it true. 1025. In point makes conditions true, e.g., 1028. Out point makes the condition false, e.g., 512.

Exercise 7

We should always test the behavior of our program when any expected data actually does not exist (EXISTENCE).

Structural-based testing

Exercise 1

Example of a test suite that achieves 100% line coverage:

```java
@Test
public void removeNullInListTest() {
    LinkedList<Integer> list = new LinkedList<>();

    list.add(null);

    assertTrue(list.remove(null));
}

@Test
public void removeElementInListTest() {
    LinkedList<Integer> list = new LinkedList<>();

    list.add(7);

    assertTrue(list.remove(7));
}

@Test
public void removeElementNotPresentInListTest() {
    LinkedList<Integer> list = new LinkedList<>();

    assertFalse(list.remove(5))
}
```

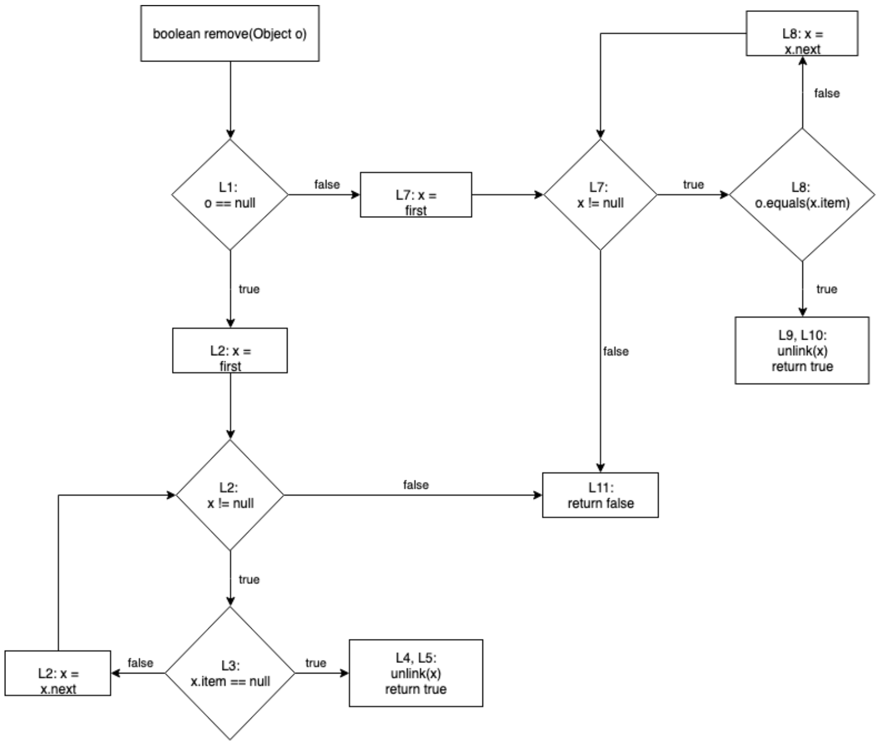
Note that there exists a lot of test suites that achieve 100% line coverage, this is just an example.

You should have 3 tests. At least one test is needed to cover lines 4 and 5 (`removeNullInListTest` in this case). This test will also cover lines 1-3.

Then a test for lines 9 and 10 is needed (`removeElementInListTest`). This test also covers lines 6-8.

Finally a third test is needed to cover line 11 (`removeElementNotPresentInListTest`).

Exercise 2



L\ in the diagram represents the line number of the code that is in the block or decision.

Exercise 3

Option 1 is the false one.

A minimal test suite that achieves 100\% (either basic or full) condition has the same number of tests as a minimal test suite that achieves 100\% branch coverage. All decisions have just a single branch, so condition coverage doesn't make a difference here. Moreover, a test case that exercises lines 1, 6, 7, 8, 9, 10 achieves around 54\% coverage (6/11).

Exercise 4

Example of a test suite that achieves 100% branch coverage:

```
@Test
public void removeNullAsSecondElementInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  list.add(5);
  list.add(null);

  assertTrue(list.remove(null));
}

@Test
public void removeNullNotPresentInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  assertFalse(list.remove(null));
}

@Test
public void removeElementSecondInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  list.add(5);
  list.add(7);

  assertTrue(list.remove(7));
}

@Test
public void removeElementNotPresentInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  assertFalse(list.remove(3));
}
```

This is just one example of a possible test suite. Other tests can work just as well. You should have a test suite of 4 tests.

With the CFG you can see that there are decisions in lines 1, 2, 3, 7 and 8. To achieve 100% branch coverage each of these decisions must evaluate to true and to false at least once in the test suite.

For the decision in line 1, we need to remove `null` and something else than `null`. This is done with the `removeElement` and `removeNull` tests.

Then for the decision in line 2 the node that `remove` is looking at should not be null and null at least once in the tests. The node is `null` when the end of the list had been reached. That only happens when the element that shouls be removed is not in the list. Note that the decision in line 2 only gets executed when the element to remove is `null`. In the tests, this means that the element should be found and not found at least once.

The decision in line 3 checks if the node thet the method is at now has the element that should be deleted. The tests should cover a case where the element is not the item that has to be removed and a case where the element is the item that should be removed.

The decisions in lines 7 and 8 are the same as in lines 2 and 3 respectively. The only difference is that lines 7 and 8 will only be executed when the item to remove is not `null`.

Exercise 5

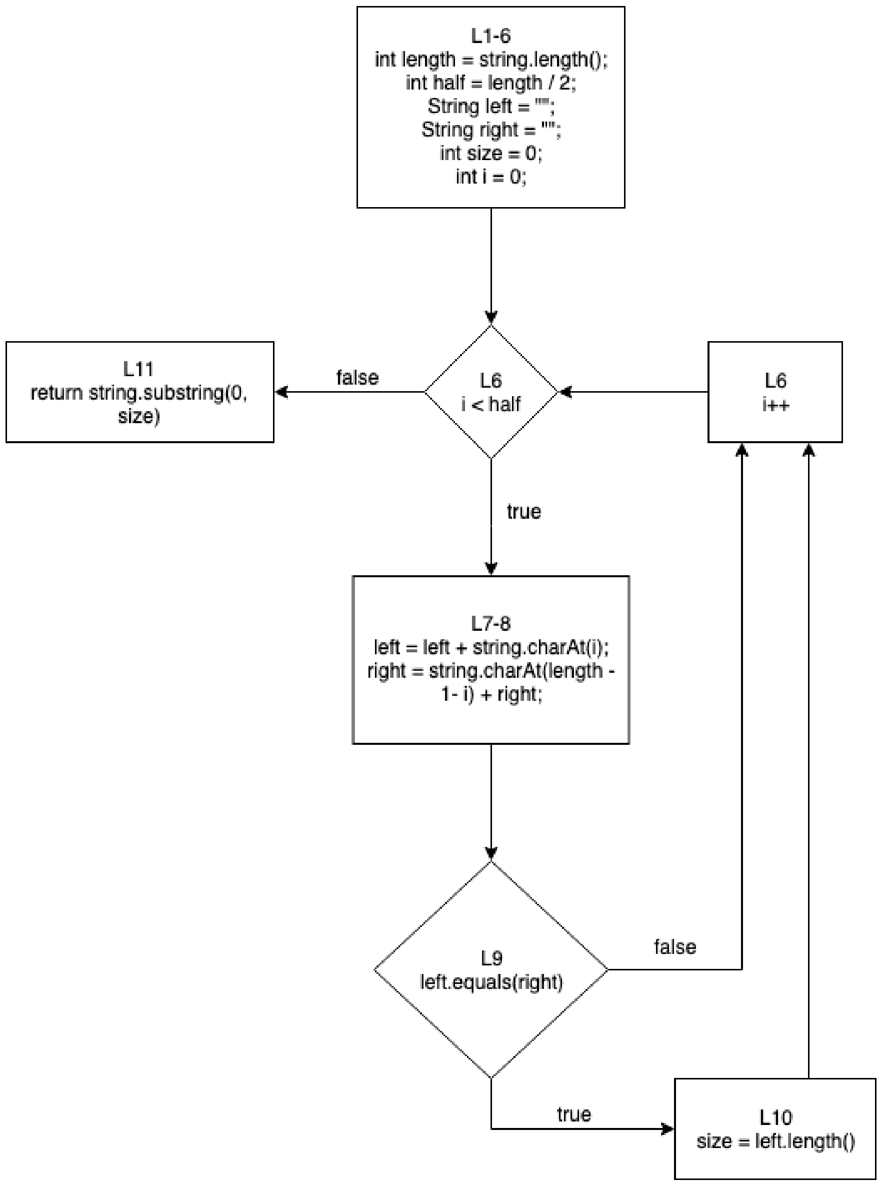
First, we find the pairs of tests that can be used for each of the conditions:

- A: {2, 6}
- B: {1, 3}, {2, 4}, {5, 7}
- C: {5, 6}

For A and C we need the decisions 2, 5 and 6. Then you can choose to add either 4 or 7 to cover condition B.

The possible answers are: {2, 4, 5, 6} or {2, 5, 6, 7}.

Exercise 6



L\ represents the line numbers that the code blocks cover.

Exercise 7

A lot of input strings give 100% line coverage. A very simple one is `"aa"`. As long as the string is longer than one character and makes the condition in line 9 true, it will give 100% line coverage. For `"aa"` the expected output is `"a"`.

Exercise 8

Answer 4. is correct. The loop in the method makes it impossible to achieve 100% path coverage. This would require us to test all possible number of iterations. For the other answers we can come up with a test case: `"aXYa"`

Exercise 9

First the condition coverage. We have 8 conditions:

1. Line 1: `n % 3 == 0`, true and false
2. Line 1: `n % 5 == 0`, true and false
3. Line 3: `n % 3 == 0`, true and false
4. Line 5: `n % 5 == 0`, true and false

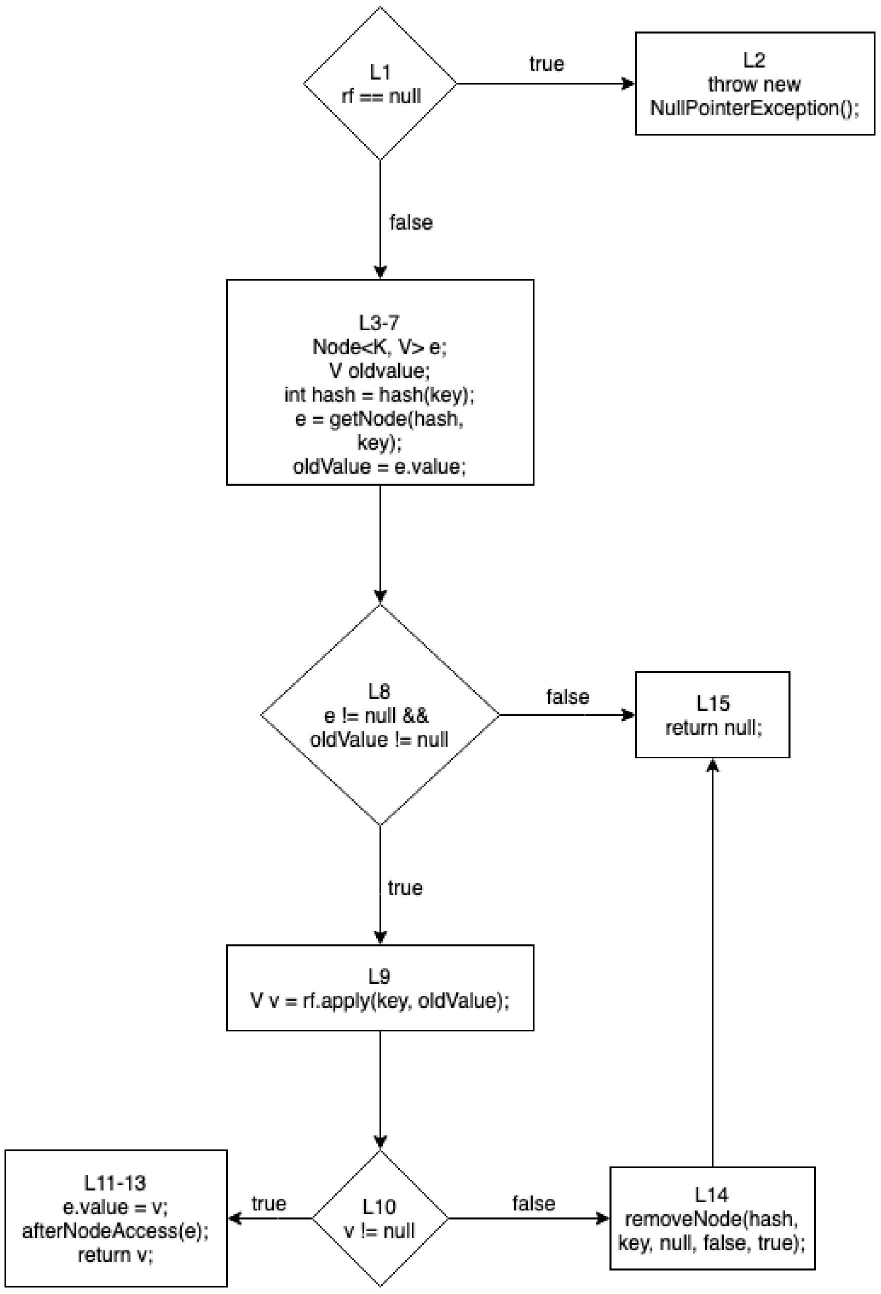
T1 makes conditions 1 and 2 true and then does not cover the other conditions. T2 makes all the conditions false. In total these test cases then cover $2 + 4 = 6$ conditions so the condition coverage is $\frac{6}{8} \cdot 100\% = 75\%$

Now the decision coverage. We have 6 decision:

1. Line 1: `n % 3 == 0 && n % 5 == 0`, true and false
2. Line 3: `n % 3 == 0`, true and false
3. Line 5: `n % 5 == 0`, true and false

Now T1 makes decision 1 true and does not cover the other decisions. T2 makes all the decision false. Therefore the coverage is $\frac{4}{6} \cdot 100\% = 66\%$.

Exercise 10



The L\ in the blocks represent the line number corresponding to the blocks.

Exercise 11

Answer: 3.

One test to cover lines 1 and 2. Another test to cover lines 1, 3-7 and 8-13. Finally another test to cover lines 14 and 15. This test will also automatically cover lines 1, 3-10.

Exercise 12

Answer: 4.

From the CFG we can see that there are 6 branches. We need at least one test to cover the true branch from teh decision in line 1. Then with another test we can cover false from L1 and false from L8. We add another test to cover false from the decision in line 10. Finally an additional test is needed to cover the true branch out of the decision in line 10. This gives us a minimum of 4 tests.

Exercise 13

MC/DC does subsume statement coverage. Basic condition coverage does not subsume branch coverage; full condition coverage does.

Exercise 14

Option 1 is correct.

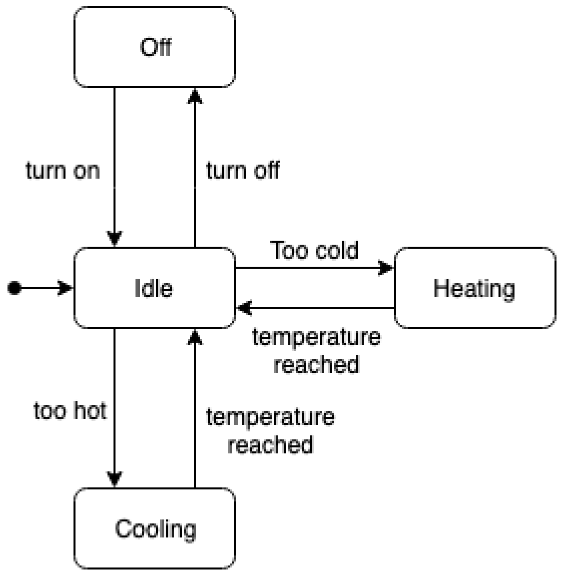
Exercise 15

Option 1 is the correct one.

Tests for A = (2,6), B = (2,4), C = (3, 4), (5, 6), (7,8). Thus, from the options, tests 2, 3, 4 and 6 are the only ones that achieve 100% MC/DC. Note that 2, 4, 5, 6 could also be a solution.

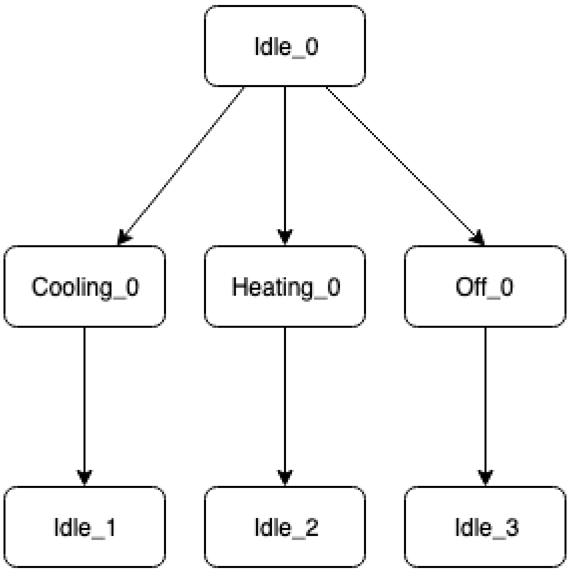
Model-Based Testing

Exercise 1



You should not need more than 4 states.

Exercise 2

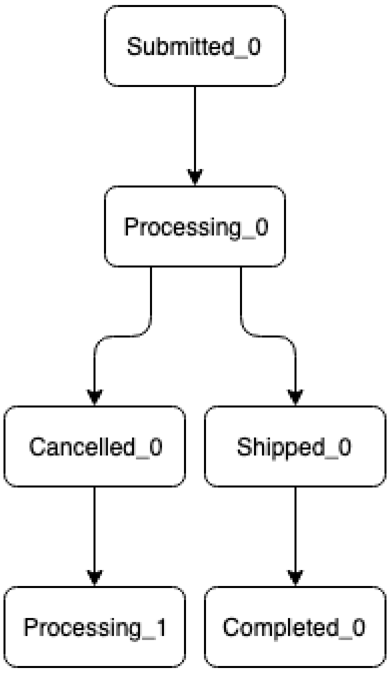


Exercise 3

	temperature reached	too hot	too cold	turn on	turn off
Idle		Cooling	Heating		Off
Cooling	Idle				
Heating	Idle				
Off				Idle	

There are 14 empty cells in the table, so there are 14 sneaky paths that we can test.

Exercise 4



Exercise 5

We have a total of 6 transitions. Of these transitions the four given in the test are covered and order cancelled and order resumed are not. This coves a transition coverage of $\frac{4}{6} \cdot 100\% = 66.7\%$

Exercise 6

STATES	Events				
	Order received	Order cancelled	Order resumed	Order fulfilled	C d
Submitted	Processing				
Processing		Cancelled		Shipped	
Cancelled			Processing		
Shipped					C
Completed					

Exercise 7

Answer: 20.

There are 20 empty cells in the decision table.

Also we have 5 states. This means $5 \cdot 5 = 25$ possible transitions. The state machine gives 5 explicit transitions so we have $25 - 5 = 20$ sneak paths.

Exercise 8

First we find pairs of decisions that are suitable for MC/DC: (We indicate a decision as a sequence of T and F. TTT would mean all conditions true and TFF means C1 true and C2, C3 false)

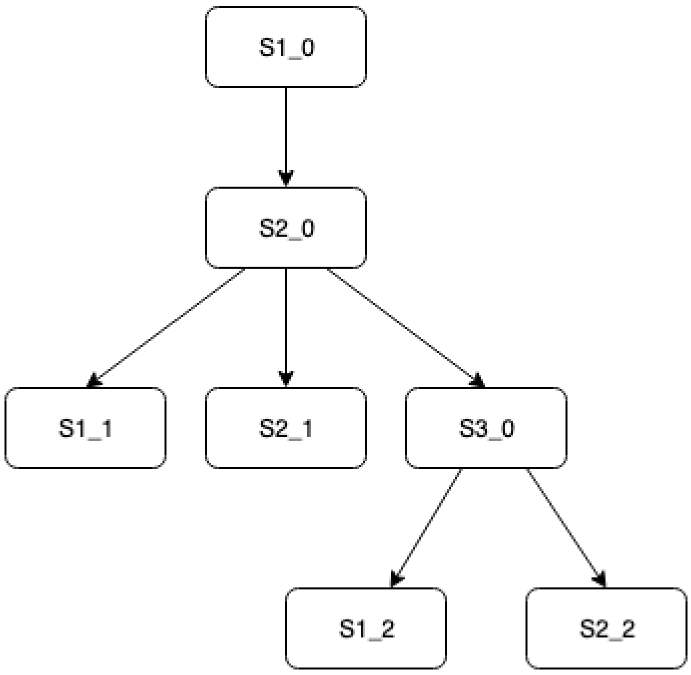
- C1: {TTT, FTT}, {FTF, TTF}, {FFF, TFF}, {FFT, TFT}
- C2: {TTT, TFT}, {TFF, TTF}
- C3: {TTT, TTF}, {FFF, FFT}, {FTF, FTT}, {TFF, TFT},

All condition can use the TTT decision, so we will use that. Then we can add FTT, TFT and TTF. Now we test each condition individually with it changing the outcome.

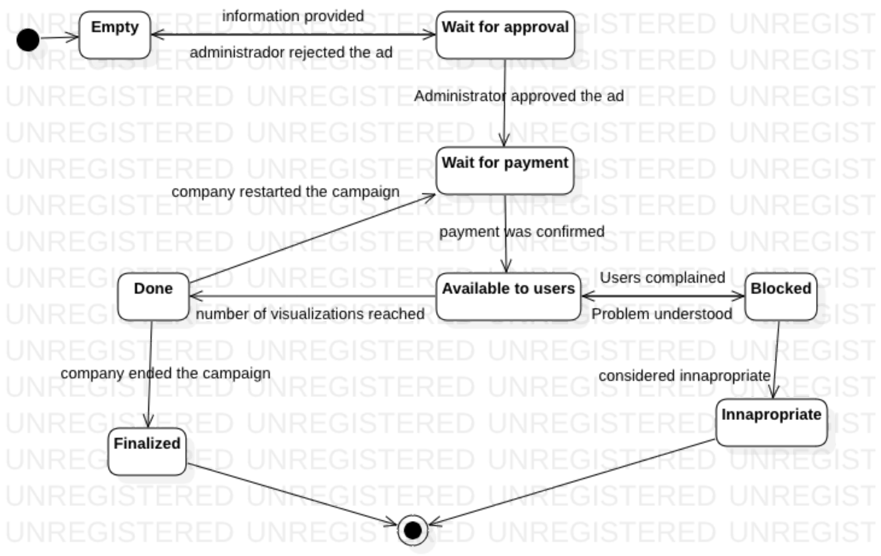
It might look line we are done, but MC/DC requires each action to be covered at least once. To achieve this we add the FFF and TFF decision as test cases.

In this case we need to test each explicit decision in the decision table.

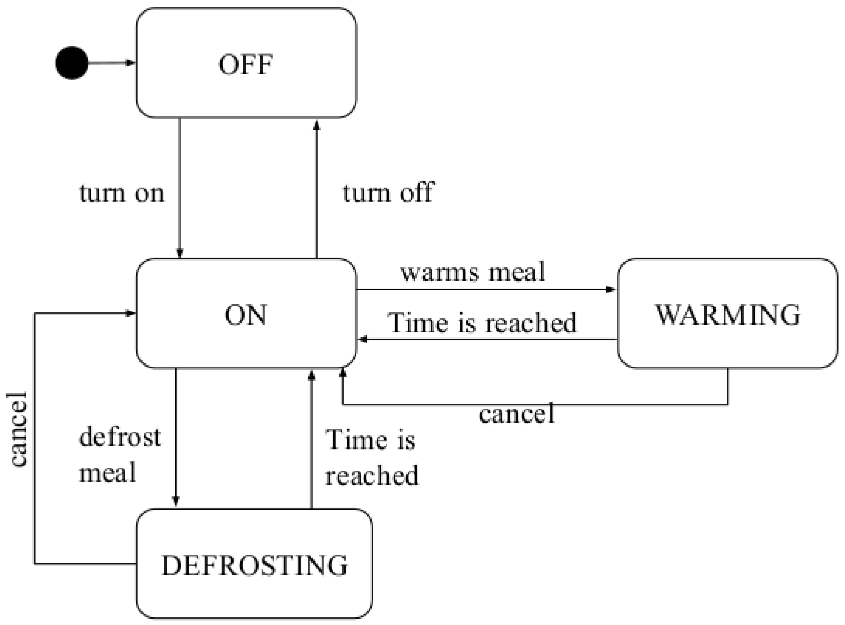
Exercise 9



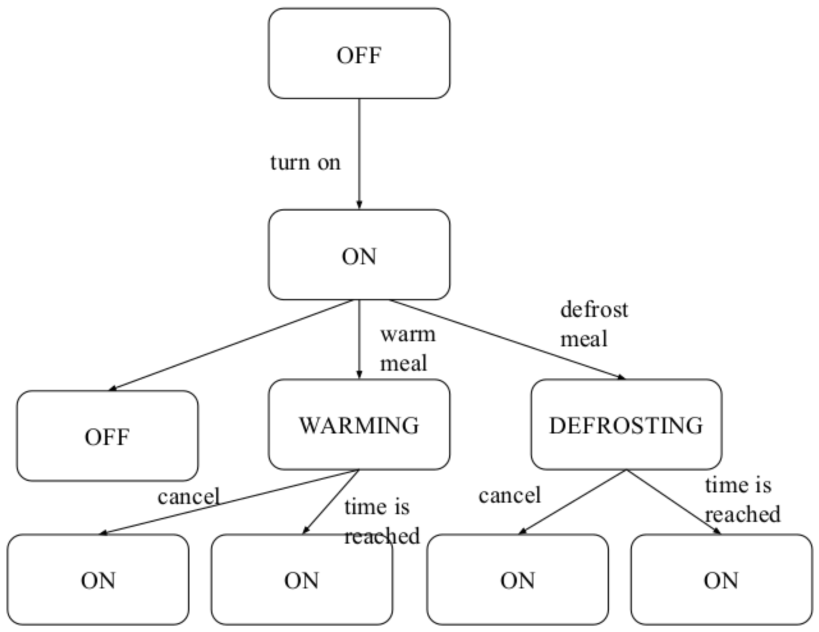
Exercise 10



Exercise 11



Exercise 12



Exercise 13

There will be one extra super state (ACTIVE), which will be a superstate of the existing WARMING and DEFROSTING states. The edges from ON to WARMING and DEFROSTING will remain. The two (cancel and time out) outgoing edges from WARMING and DEFROSTING (four edges in total) will be replaced by two edges going out of the super ACTIVE state. So there will be two fewer transitions.

Exercise 14

	C1	C2	C3	C4	C5	C6	C7
Valid format?	T	T	T	T	F	F	F
Valid size?	T	T	F	F	T	T	F
High resolution?	T	F	T	F	T	F	T
Outcome	success	success	fail	fail	fail	fail	fail

Exercise 15

	T1	T2	T3
User active in past two weeks	T	T	T
User has seen ad in last two hours	F	F	F
User has over 1000 followers	T	F	F
Ad is highly relevant to user	T	T	F
Serve ad?	T	T	T

Design-by-contracts and property-based testing

Exercise 1

`board != null`

For a class invariant the assertions has to assert a class variable. `board` is such a class variable, unlike the other variables that are checked by the assertions. The other assertions are about the parameters (preconditions) or the result (postcondition).

Exercise 2

The existing preconditions are **not** enough to ensure the property in line 10.

`board` itself cannot be `null` and `x` and `y` will be in its range, but the content of board can still be `null`. To guarantee the property again the method would have to implicitly assume an invariant, that ensures that no place in `board` is `null`.

In order to do this, we would have to make the constructor ensure that no place in `board` is `null`. So we have to add an assertion to the constructor that asserts that every value in `board` is not `null`.

Exercise 3

The second colleague is correct.

There is a problem in the `Square`'s preconditions. For the `resetSize` method these are stronger than the `Rectangle`'s preconditions. We do not just assert that the `width` and `height` should be larger than 0, but they should also be equal.

This violates the Liskov's Substitution Principle. We cannot substitute a `Square` for a `Rectangle`, because we would not be able to have unequal width and height anymore.

Exercise 4

Making correct use of a class should never trigger a class invariant violation. We are making correct use of the class, as otherwise it would have been a precondition violation. This means that there is a bug in the implementation of the library, which would have to be fixed. As this is outside your project, you typically cannot fix this problem.

Exercise 5

Just like the contracts we have a client and a server. \ A 4xx code means that the client invoked the server in a wrong way, which corresponds to failing to adhere to a precondition. \ A 5xx code means that the server was not able to handle the request of the client, which was correct. This corresponds to failing to meet a postcondition.

Exercise 6

P' should be equal or weaker than P, and Q' should be equal or stronger than Q.

Exercise 7

To make debugging easier.

Testing pyramid

Exercise 1

1. Manual
2. System
3. Integration
4. Unit
5. More reality (interchangeable with 6)
6. More complexity (interchangeable ith 5)

See the diagram in the Testing Pyramid section.

Exercise 2

The correct answer is 1.

1. This is correct. The primary use of integration tests is to find mistakes in the communication between a system and its external dependencies
2. Unit tests do not cover as much as integration tests. They cannot cover the communication between different components of the system.
3. When using system tests the bugs will not be easy to identify and find, because it can be anywhere in the system if the test fails. Additionally, system tests want to execute the whole system as if it is run normally, so we cannot just mock the code in a system test.

4. The different test levels do not find the same kind of bugs, so settling down on one of the levels is not a good idea.

Exercise 3

Option 4 is not required.

Changing the transaction level is not really required. Better would be to actually exercise the transaction policy your application uses in production.

Exercise 4

Correct answer: Transitioning from a testing pyramid to an ice-cream cone anti-pattern

Exercise 5

Unit testing.

Exercise 6

The interaction with the system is much closer to reality.

Exercise 7

System tests tend to be slow and often are non-deterministic. See https://martinfowler.com/bliki/TestPyramid.html!

Mock Objects

Exercise 1

The correct answer is 4.

1. This line is required to create a mock for the `OrderDao` class.
2. With this line we check that the methods calls start with `order` on a `delivery` mock we defined. The method is supposed to start each order that is paid but not delivered.
3. With this line we define the behavior of the `paidButNotDelivered` method by telling the mock that it should return an earlier defined `list` .
4. We would never see this happen in a test that is testing the `OrderDeliveryBatch` class. By mocking the class we do not use any of its implementation. But the implementation is the exact thing we want to test. In general we never mock the class under test.

Exercise 2

You need mocks to both control and observe the behavior of the (external) conditions you mocked.

Exercise 3

Option 1 is the false one. We can definitely get to 100% branch coverage there with the help of mocks.

Exercise 4

Only approach 2.

Design for Testability

Exercise 1

To test just the `runBatch` method of `OrderDeliveryBatch` (for example in a unit test) we need to be able to use mocks for at least the `dao` and `delivery` objects. In the current implementation this is not possible, as we cannot change `dao` or `delivery` from outside. In other words: We want to improve the controllability to improve the testability.

The technique that we use to do so is called dependency injection. We can give the `dao` and `delivery` in a parameter of the method:

```
public class OrderDeliveryBatch {

  public void runBatch(OrderDao dao, DeliveryStartProcess delivery) {
    List<Order> orders = dao.paidButNotDelivered();

    for (Order order : orders) {
      delivery.start(order);

      if (order.isInternational()) {
        order.setDeliveryDate("5 days from now");
      } else {
        order.setDeliveryDate("2 days from now");
      }
    }
  }
}
```

Alternatively we can create fields for the `dao` and `delivery` and a constructor that sets the fields:

```
public class OrderDeliveryBatch {

  private OrderDao dao;
  private DeliveryStartProcess delivery;

  public OrderDeliveryBatch(OrderDao dao, DeliveryStartProcess delivery) {
    this.dao = dao;
    this.delivery = delivery;
  }

  public void runBatch() {
    List<Order> orders = dao.paidButNotDelivered();

    for (Order order : orders) {
      delivery.start(order);

      if (order.isInternational()) {
        order.setDeliveryDate("5 days from now");
      } else {
        order.setDeliveryDate("2 days from now");
      }
    }
  }
}
```

Exercise 2

The method and class lack controllability. We cannot change the values that `Calender` gives in the method because the `getInstance` method is static. Mockito cannot really mock static methods, which is why we tend to avoid using static methods.

We can use depencency injection to make sure we can control the `today` object by using a mock.

Exercise 3

The correct answer is 1 and 3.

As we discussed it is very important to keep the domain and infrastructure separated for the testability. This can be done, for example, by using Ports and Adapters.

Static methods cannot be mocked and are therefore very bad for the controllability of the code. Code that has low controllability also has a low testability, so replacing the static methods by non-static ones will be very beneficial to the testability.

The large tables and lack of indices do not really influence the testability, especially not when talking about unit tests. There we end up mocking the classes interacting with the database anyway.

Too many attributes/fields can hurt testability as we might need to create a lot of mocks for just one class under test. However, the static methods and mixed domain and infrastructure are worse for the testability than a large amount of attributes/fields.

Exercise 4

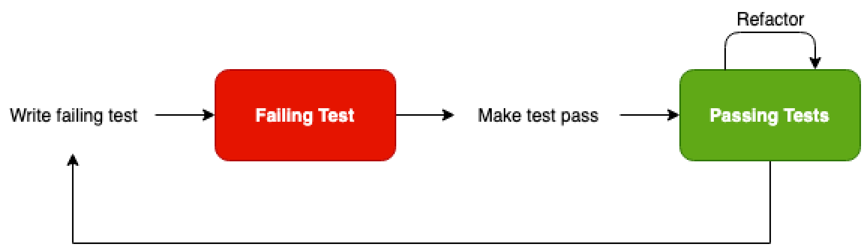
1. Observability: The developer needs to be able to better observe the result.
2. Controllability: The developer has to be able to change (control) a certain variable or field.
3. Controllability: The developer should be able to control what instance of a class the class under test uses.

Test-Driven Development

Exercise 1

1. Write failing test
2. Failing test
3. Make test pass
4. Passing test
5. Refactor

From the explanation above:



Exercise 2

How did it feel to practice TDD?

Exercise 3

Option 1 is the least important one.

Although a few studies show that the number of tests written by TDD practitioners are often higher than the number of tests written by developers not practicing TDD, this is definitely not the main reason why developers have been using TDD. In fact, among the alternatives, it's the least important one. All other alternatives are more important reasons, according to the TDD literature (e.g., Kent Beck's book, Freeman's and Pryce's book.

Exercise 4

Option 1 is not a benefit of TDD. The TDD literature says nothing about team integration.

Test code quality

Exercise 1

Correct: Both tests are very slow.

Exercise 2

Correct answer: Mystery guest

Exercise 3

Correct answer: It is hard to tell which of several assertions within the same test method will cause a test failure.

Exercise 4

Flaky test.

Exercise 5

To avoid the flakiness, a developer could have mocked the random function. It does not make sense, the test is about testing the generator and its homogeinity; if we mock, the test looses its purposes.

Mutation testing

Exercise 1

Mutation testing.