Chapter 5

Test-Driven Development

Learning Objectives

Extreme Programming is an agile process for developing software fast. In this chapter the learning objective is to demonstrate a central XP technique, namely *test-driven development*. Test-driven development has a strong focus on crafting reliable and maintainable code: *Clean code that works*. Test cases demonstrate that it *works* while you also constantly restructure your code to make it *clean*.

In this chapter

- You will become familiar with the *rhythm* of test-driven development that drives each iteration, iterations that each have a specific goal namely to add a single feature to the production code.

- You will learn a set of *principles* that form a list of potential actions you can take in each step in the rhythm.

- You will learn the definition of *refactoring* and see some simple example of refactoring the production and testing code.

- You will learn about the liabilities and benefits of a test-driven development process.

I will demonstrate these techniques in practice as I need them to implement the pay station system. Test-driven development is not a question of being able to recite a long list of principles and definitions that you have learned by heart, it is a question of being able to apply them in pratice in your program development. Therefore this chapter devotes quite a lot of space to the *devils in the details* at the code level.

5.1 Values of Test-Driven Development

Test-driven development, referred to as *TDD*, is an intrinsic part of *Extreme Programming* that was outlined in Chapter 1. A central point in agile methods is *working software* and TDD most certainly focuses on producing reliable and maintainable software while programming at maximal speed.

There has been a tendency for traditional software development thinking that fixing defects late in a software product's development lifetime is extremely expensive and thus enormous emphasis has been put on the early analysis and design phases. Implementation and coding were seen as "merely" translating the finished design into executable software. Failures in sofware were often seen as indications that not enough time had been spent on the design. Many seasoned software developers, however, know that this is not the whole truth. Software coding is a learning process that is central in getting to understand both the problem and solution domain. Coding is for software design what experiments are for natural science: A way to ground hypotheses in reality. This does not mean that you should not design—the point is that designs should not stay too long in your head or on your desk before their validity and soundness are tested as computer programs.

This leads to an important point, **speed**. You strive for developing the production code as fast as possible without compromising quality. Speed is not achieved by being sloppy or by rushing, if you want high quality code. On the contrary it is achieved by having a well structured programming process (the "rhythm") guided by sound principles, by testing design ideas early, and by keeping the code maintainable and of high quality. Another important means to speed is **simplicity** as expressed in one of the mantras of agile development: "Simplicity–the art of maximizing the amount of work not done–is essential." In TDD you implement the code that makes the product work, and no more.

Another central point or value in TDD is to **take small steps**. If a story for the system requires you to add a new class, make relations to it from existing classes, and modify the object instantiation sequence, then you have three steps—at the very least. TDD puts extreme emphasis on taking one, very tiny, step at the time; even if it means writing code that will be removed already in the next step. Why? Because when you try to leap over several steps you usually find yourself falling into an abyss of problems. Before I learned TDD, I have several times found myself in a software quagmire because I have tried to implement a complex feature in a single step. The story usually went along these lines: *After implementing a small parts of the feature, I found that some of the implemented functionality was similar to something in another class, therefore I started changing that class so I could use it instead. During that work I discovered a defect that I then began to fix. However, in order to fix it I had to change the internal data structure quite a lot which lead to adding a parameter to a few of the class' methods. Therefore I had to change all the places where these methods were called. During that work I notice that . . .* After a few days I had made changes in so many places that the system stopped working and I had lost track of why I started all this work in the first place! More often than not I simply had to undo all the changes to get back on track and I learned the value of taking backups very often! In TDD you concentrate on one step at the time—even if it means writing code that will only be used temporarily . Fixing one step and making it work gives confidence and it is easy to return to safe ground if the ideas for the next step turn out to be bad.

A final value is to **keep focus**. You must focus on one step, one issue, at the time. Otherwise code has a fantastic way of detracting your attention so often that you quickly forget the point of the whole exercise. Just look at the story above. So, in my development effort below I will *take small steps* and *keep focus*.

5.2 Setting the Stage

So, I have the specification of the pay station system and some Java interfaces (Chapter 4), I have a computer with some development tools like an editor and a compiler, and I have a need to produce a implementation that satisfies the requirements. As stated in the beginning, I also have a wish not just for any implementation but for a *reliable* implementation. Thus, I need to go from A to B, and TDD is a process that helps me by providing a path between the two points. TDD does so by listing a number of **TDD principles**, strategies or rules that I use to decide what to do next in my programming in order to keep focus and take small steps. The most important rule of them all has given the technique its name:

TDD Principle: **Test First**

When should you write your tests? Before you write the code that is to be tested.

This principle is perhaps a rather surprising statement at first. How do I write the test first, it must mean that I test something that I have not written yet? How do I test something that does not exist? The consequence has historically been that naturally the production code was written first. However, history has also shown that in almost all cases the tests have never been written. I have seen the force to postpone writing tests many times in my own professional life: *"I ought to write some tests, but I feel in a hurry, and anyway it appears to work all right when I run it manually."* However, a few months of working this way and, *Bang*, weird failures begins to surface. Now the defects are tricky to track down because I do not have the small tests that verify the behavior of each small part of the software.

TDD Principle: **Automated Test**

How do you test your software? Write an automated test.

In TDD you run *all* the tests *all* the time to ensure that your last modifications have not accidentally introduced a defect. Therefore manual tests are inadequate because they are too labor intensive. A good unit testing tool is therefore essential, so I ready JUnit (see Section 2.4) for the job of writing my automated tests.

So, I will start the implementation of the pay station by writing JUnit tests and by writing them first. But how do I structure this process? A simple suggesting is:

TDD Principle: **Test List**

What should you test? Before you begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests.

The test list is a sensible compromise between "moving fast" and "keeping focus." I want to start as soon as possible but I need to maintain a keen eye on what I want to achieve and what the results are. A test list is like a shopping list for the grocery store. It keeps me focused, I don't forget a lot of things, but it does not exclude me from putting some extra stuff in the shopping bag.

Thus, looking over the three stories I developed together with the customer in the previous chapter: *buy a parking ticket*, *cancel a transaction*, and *reject illegal coin*, a first test list may look like this:

- ✻ accept legal coin
- ✻ 5 cents should give 2 minutes parking time
- ✻ reject illegal coin
- ✻ readDisplay
- ✻ buy produces valid receipt
- ✻ cancel resets pay station

If you review the stories you may come up with other suggestions for a test list or may find that this one is too simple. However, the point is to start moving forward fast: arguing over the "best" list does not produce software, it only delays the time when we start producing it. Many issues will automatically arise once we get started implementing test cases and production code. As we program we gain a deeper understanding of the problem and the test list is something that we constantly add to and refine just as a grocery list.

The TDD process consists of five steps that you repeat over and over again: The *TDD rhythm.* The five steps are:

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Ideally you try to move through these five steps as quickly as possible while keeping your focus on the implementation job to be done.

5.3 Iteration 1: Inserting Five Cents

Okay, I have the test list with several features to implement (or more correctly: tests of features to write before implementing the features), I have the rhythm, the question remaining is *which test to write first?*

TDD Principle: **One Step Test**

Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

Looking over the list I see some features that are obvious, like for instance *readDisplay* which is probably just returning some instance variable. I do not learn much from choosing that item from the list. A test list often also contains items where I have no clue at all as how to implement. While there are no obviously highly complex items on our pay station test list, still the "buy" item is more complicated as it seems to require several features to be working first. These items are not good candidates either—postpone them until later where your work with the implementation has taught you more about the system and you can rely on more functionality to be present. The *One Step Test* principle tells me to pick a feature that is not too obvious but that I am confident I can implement and thus learn from: I take *one step* forward. Looking over the test list I decide to go for the test "5 cents = 2 minutes parking:" Not too obvious nor too complex.

Step 1: Quickly add a test. To paraphrase this statement in our testing terminology from Chapter 2, I need to write a test case in which the unit under test is the implementation of the pay station in general and the addPayment method in particular; the input value is 5 cents and the expected output is that we can read 2 minutes parking time as the value to be displayed. In JUnit the test case can be expressed like this.

Listing: chapter/tdd/iteration-1/TestPayStation.java

```java
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

Sidebar 5.1 explains how you download, compile, and run the code associated with this chapter.

☞ Study and run the code as you read through this chapter.

Step 2: Run all tests and see the new one fail. We start by compiling production and test code but, of course, the compilation fails! I have not yet provided any class that implements the PayStation interface. Now, I am tempted to start implementing the behavior I want, but TDD tells me to *take small steps.* Therefore, I implement the smallest possible implementation that can compile, even though it is presently functionally incomplete. In practice, it means that all method bodies are empty except those that have a return value: these return 0 or null. Such a minimal implementation is often called a **temporary stub** and in TDD any newly introduced class starts its life as a temporary stub. I will explore and extend the stub concept in detail in Chapter 12.

Sidebar 5.1: Using the Pay Station Code

The source code as well as compilation scripts for Windows and Linux are provided at the book's web site (http://www.baerbak.com). You need to download the zip archive, unzip it and then you will be able to find the source code in the folders listed above each source code listing.

There is a folder for each iteration in this chapter. The contents of the folder shows the production and testing code at the *end* of the iteration. Thus the folder *chapter/tdd/iteration-1* contains the code just before iteration 2 starts. There is also a *iteration-0* folder containing the code before the first iteration.

Each directory contains a "compile" script as well as a "run-test" script for respectively compilation and execution of the test cases (.bat for Windows and .sh for Bash shell on Linux). The JUnit jar file is provided in the source folders so there is no installation of JUnit involved. For a discussion of the contents of the scripts, please refer to sidebar 2.3 on page 22.

Listing: chapter/tdd/iteration-0/PayStationImpl.java

```java
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 0;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

Now, at least, I can compile and is ready to run my test which outputs:

```
JUnit version 4.4
.E
Time: 0,047
There was 1 failure:
1) shouldDisplay2MinFor5Cents(TestPayStation)
java.lang.AssertionError: Should display 2 min for 5 cents
  expected:<2> but was:<0>
     at org.junit.Assert.fail(Assert.java:74)
     at org.junit.Assert.failNotEquals(Assert.java:448)
     at org.junit.Assert.assertEquals(Assert.java:102)
     at org.junit.Assert.assertEquals(Assert.java:323)
     at TestPayStation.shouldDisplay2MinFor5Cents(Test
                                               PayStation.java:20)
  [lines removed here]
```

```
     at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

FAILURES!!!
Tests run: 1,  Failures: 1
```

Returning to our case study, the requirements speak of two minutes parking time for five cents but the production code returns zero. This is hardly surprising considering that our initial method implementation simply returns 0.

Step 3: Make a little change, but what change should I make? TDD provides a principle that always upsets seasoned software developers:

TDD Principle: **Fake It ('Til You Make It)**

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

What? I have to write code that is obviously incomplete? Yes, this principle may seem absurd at first. I will discuss it at length below and as we proceed with the pay station implementation but for now let us just use it. Applying this principle is of course easy—I simply return a constant in the appropriate method in PayStationImpl:

```
  public int readDisplay() {
    return 2;
  }
```

Step 4: Run all tests and see them all succeed.

```
JUnit version 4.4
.
Time: 0,016

OK (1 test)
```

The passed testcase marks success, progress, and confidence!

Looking over the test list again, I see that I have actually tested quite a few of our initial test cases.

- ~~accept legal coin~~
- ~~5 cents should give 2 minutes parking time.~~
- reject illegal coin
- ~~readDisplay~~
- buy produces valid receipt
- cancel resets pay station

However, you may be concerned at this stage. Our test case passes but surely the production code is incomplete! It will only pass this single test case! And I have apparently wasted time and effort on writing clearly incomplete code that needs to be changed in just a few minutes. This is insane, is it not?

Actually, there are good reasons in this apparent madness. First, the focus of this iteration is *not* to implement a full rate calculation but simply to handle the *5 cents = 2 minutes* entry on the test list. Thus, this *is* the correct and smallest possible implementation of this single test case. Remember the technique is called test-*driven*. If I had begun implementing a complete calculation algorithm then this production code would not have been *driven* into existence by my test cases! This is a key point in TDD:

Key Point: Production code is driven into existence by tests

In the extreme, you do not enter a single character into your production code unless there is a test case that demands it.

This leads to the second point, namely that of course the *Fake It* principle can not stand alone, it must be considered in conjunction with the *Triangulation* principle:

TDD Principle: **Triangulation**

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

☞ Use Wikipedia or another internet resource to find out what triangulation is in ordinary geometry.

In TDD *Triangulation* states that you must have two, three, or more examples in order to generalize, to *drive an abstraction* like an algorithm or a class into existence. If I had simply started implementing the algorithm for parking time calculations based only on our one test case, odds are that I would have missed something. (Of course, our case is pretty simple here but the principle gets much more valuable when the algorithms are more complex.)

So, I know that I have just introduced *Fake It* code and I know I have to triangulate it away in one of the following iterations. How do I make sure I remember this? This is where the test list comes to my rescue: I simply add a new test case to the test list:

- ~~accept legal coin~~
- reject illegal coin, exception
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- buy produces valid receipt
- cancel resets pay station
- *25 cents = 10 minutes*

In this iteration 1, the *Fake It* principle has helped me to *keep focus* and *take small steps*! Still, this principle appears odd as first sight, and you need to have applied it a number of times to really appreciate it.

Step 5: Refactor to remove duplication. There appears to be no duplication, so this step is skipped in this iteration.

5.4 Iteration 2: Rate Calculation

Step 1: Quickly add a test. The last iteration left us with production code containing *Fake It* code. Such code should not stay for long and it should certainly not accumulate. I therefore choose the *25 cents = 10 minutes* item as the focus for my next iteration.

The question is how I should express this test case. Two paths come to my mind. One path is to just add more test code into the existing @Test method to produce something like

```
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );

    ps.addPayment( 25 );
    assertEquals( "Should display 12 min for 30 cents",
                  12, ps.readDisplay() );
  }
```

The second path is to create a new @Test method in the test class. TDD has a strong opinion concerning the amount of testing done in each method:

TDD Principle: **Isolated Test**

How should the running of tests affect one another? Not at all.

This principle, taken literally, tells me to make another test case to avoid interference between our two test cases. If you first insert 5 cents, and next want to test the amount of parking time given for 25 cents you must test that the display reads 12 minutes because the total amount entered is 30 cents. This interference between the test cases is relatively easy to overlook here, but in the general case things can get complex and you quickly end up in a **ripple effect** in the test cases: it is like dominos—if one falls then they all fall. To see this, consider a situation later in development where some developer accidently introduces a defect so the pay station cannot accept 5 cent coins. In the *Isolate Test* case, then the 5 cent test case would fail and the 25 cent test case would pass. This is a clear indication of what defect has been introduced. In the case where both test cases are expressed in the same test method, however, the whole test method fails. Thus the problems appears worse than it actually is (because apparently both 5 cent and 25 cent entry fails) and also the problem is more difficult to identify (is it a defect in the 5 cent or in the 25 cent validation code?)

Back to our pay station. I write an isolated test for 25 cent parking.

```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

Step 2: Run all tests and see the new one fail. I get the expected failed tests:

```
1) shouldDisplay10MinFor25Cents(TestPayStation)
java.lang.AssertionError: Should display 10 min for 25 cents
  expected:<10> but was:<2>
  [lines omitted]
  at TestPayStation.shouldDisplay10MinFor25Cents
                                      (TestPayStation.java:32)
  [lines omitted]
```

Step 3: Make a little change. To get the behavior I must introduce a multiplication of whatever amount has been inserted so far. This is not really rocket science, so I come up with:

Fragment: chapter/tdd/iteration-2/PayStationImpl.java

```
1 public class PayStationImpl implements PayStation {
2   private int insertedSoFar;
3   public void addPayment( int coinValue )
4           throws IllegalCoinException {
5     insertedSoFar = coinValue;
6   }
7   public int readDisplay() {
8     return insertedSoFar / 5 * 2;
9   }
```

However, do you notice that this code is also incomplete? As I write the code I realize that I only have test cases that insert a single coin, not two or more. Thus the correct implementation of line 5:

```
    insertedSoFar += coinValue;
```

is not driven by any tests yet. I note that I can drive the "+=" by adding yet another item, like *enter two or more legal coins*, to my test list. But even without the summation in the addPayment method I achieve *Step 4: Run all tests and see them all succeed*. Great. Next, let us look at *Step 5: Refactor to remove duplication*. Fowler (1999) defines refactoring as:

Definition: **Refactoring**

Refactoring is the process of modifying and restructuring the source code to improve its maintainability and flexibility without affecting the system's external behavior when executing.

Thus, when I refactor, I change the code but it must behave in the exact same way before and after the code change as seen from the user or some external system. The whole point of the exercise is to "improve its maintainability and flexibility". Recall from Chapter 3 that maintainability and flexibility are qualities that speak of the cost of adding or enhancing the software's features: maintainable code is code that is easy to change. If my code contains duplicated code and the functionality it expresses has to be changed then I have to change in two places. This takes longer time of course but even worse I might forget to make the change in one of the places!

Looking over the production code there seems to be no code duplication, but there is some in the test cases—both contain the object instantiation step:

```
  PayStation ps = new PayStationImpl();
```

Merging the two test cases into one to avoid the duplication is not an option as it would not obey *Isolated Test*. Obviously, every test case I may think of will duplicate this line, so a better approach is to isolate the *"object setup"* code in a single place. In testing terminology, such a setup of objects before testing is called a **fixture**. JUnit supports fixtures directly using the @Before annotation.

Listing: chapter/tdd/iteration-2-initial/TestPayStation.java

```java
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {
  PayStation ps;
  /** Fixture for pay station testing. */
  @Before
  public void setUp() {
    ps = new PayStationImpl();
  }

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }

  /**
   * Entering 25 cents should make the display report 10 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay10MinFor25Cents()
          throws IllegalCoinException {
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  10, ps.readDisplay() );
  }
}
```

A fixture, a @Before method, is invoked once before every test method so the resulting calling sequence of the above test class will become what I need: setUp(), shouldDisplay2MinFor5Cents, setUp(), shouldDisplay10MinFor25Cents. This is the way JUnit supports the *Isolated Test* principle—each test case is independent from all others.

The last test case contains the assertion assertEquals(10, ps.readDisplay()). Where did '10' come from? It is easy to overview the calculation right now but remember that you may have to look over your test cases three months or two years from now, maybe because some refactoring has made that test case fail. At that time, the 10 may seem like a riddle. In a pressed situation with a deadline coming up, riddles are not what you want to solve, you want your code to work again! The point is yet another rule.

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

The ‘10’ is a result of a calculation and instead of giving the answer the test case communicates its intention much better by giving the calculation itself:

```
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  25 / 5 * 2, ps.readDisplay() );
    // 25 cent in 5 cent coins each giving 2 minutes parking
```

This rule especially becomes beneficial when the calculation becomes more complex than in this case. (Beware, though, of blindly copying code fragments containing the calculations from the production code into the test code or the other way around as they may both contain defects!) An alternative is to add a comment outlining the calculation.

I have now made a simple refactoring in the test code but my rhythm is not complete until JUnit has verified that I did not accidentally introduce new defects. I run JUnit again: All pass. It is a success that the new test case passes—however it is just as important that the old one does!

Key Point: All unit tests always pass

At the end of each iteration, all tests must run 100%.

Sometimes in developing more complex software you will add a new test case, modify the existing production code, only to see several other test cases suddenly fail. It is a key point in TDD that those “old” test cases *must* be made to pass before the iteration is considered finished. If you leave the old test cases failing the reliability of your software is of course quickly degrading.

Finally, I strike out the passed test on my test list:

✻ ~~accept legal coin~~
✻ reject illegal coin, exception
✻ ~~5 cents should give 2 minutes parking time.~~
✻ ~~readDisplay~~
✻ buy produces valid receipt
✻ cancel resets pay station
✻ ~~25 cents = 10 minutes~~
✻ enter two or more legal coins

5.5 Iteration 3: Illegal Coins

One Step Test tells me to pick an item from my test list that is not too simple, not to complex, and one that I can learn something from. I could go for the buy or cancel operation, or perhaps enter two coins, but I decide to look at illegal coins: my present production code contains no validation of the entered coin values. We do not have that many 17 cent coins around so let us try that. The interface specifies that an IllegalCoinException must be thrown so we must define a test case where the input value is an illegal coin and the expected output is an exception being thrown.

JUnit allows you to specify that a test should throw an exception by giving the exception class as parameter:

Fragment: chapter/tdd/iteration-3/TestPayStation.java

```
  @Test(expected=IllegalCoinException.class)
  public void shouldRejectIllegalCoin() throws IllegalCoinException {
    ps.addPayment(17);
  }
```

Step 2: Run all tests and see the new one fail. The result is of course a broken test:

```
JUnit version 4.4
...E
Time: 0,031
There was 1 failure:
1) shouldRejectIllegalCoin(TestPayStation)
java.lang.AssertionError: Expected exception: IllegalCoinException
  [lines omitted]
FAILURES!!!
Tests run: 3,  Failures: 1
```

Step 3: Make a little change. I have to introduce validation code in method addPayment. My suggestion is to use a switch to define the valid coins:

Fragment: chapter/tdd/iteration-3/PayStationImpl.java

```
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar = coinValue;
  }
```

which leads to *Step 4: Run all tests and see them all succeed.* Three tests passing. Great! So what is my present status? I have test cases for valid and invalid coins. They all pass. I do not have any *Fake It* in the production code. However, the production code is still incomplete. Hopefully you noted the missing case in the switch statement: I do not test for coin value 10 and therefore dimes are presently considered an illegal coin. This is not comparable to the specification from Alphatown. But it demonstrates that a test suite that passes is not the same as reliable software that satisfies all stories and requirements. Testing and test-driven development can never prove the absence of defects, only prove the existence of one. A short story about the truth of this is found in sidebar 5.2. The consequence is that you have to exercise much care when defining test cases so you can convince yourself and ultimately the buyer of your software that your test cases have found as many defects as possible. Yet another principle helps us:

TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.

This principle can guide my selection of data. Do not pick several data values that are essentially processed in the same manner. For instance I chose a 17 cent coin as

Sidebar 5.2: Defects in a Plotter Package

I once developed software to make graphical plots for a technical department in a major telephone company. The software operated a pen based plotter and could plot classic X-Y diagrams. As a feature I added that the creation date of the plot was written in the corner of the plot. I developed this software package for more than a week and it worked fine with all my test data.

However, two problems surfaced later. The first appeared instantly when the technical department tested it: I had tested with X-Y data in the range 0..1000 but the data the technical department needed to plot was in the 10^{-7} range and my software did not handle that well. After fixing this, all worked fine for several weeks—until the package stopped working all together. It took me quite a long time to figure out that the defect was in the code for writing the date in the corner! It turned out that I had started on the package around the 12th in the month and thus for several weeks the date integer had always consisted of two digits! As the date turned to the next month all of a sudden the date only had one digit—and my production code failed on converting the non-existing second digit into a string! Remember the *Representative Data* principle.

illegal coin, and adding ten extra test cases that try to insert a 3 cent, 42 cent, 5433 cent, etc., do not add much additional confidence in the reliability of the production code. So select a small set of data values. On the other hand your set must not miss values that *do* exhibit a computational difference. This is the case here in which the buy story states that 5, 10, and 25 cent coins are valid and my implementation using a switch treats each coin seperately. Thus I need test cases to cover all three types of coins. I remember to keep focus, this iteration is about invalid coins and it is completed, so the way to ensure I remember this observation is to add an item to the test list. In this case I can actually just change one of the existing items:

- ~~accept legal coin~~
- ~~reject illegal coin, exception~~
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- buy produces valid receipt
- cancel resets pay station
- ~~25 cents = 10 minutes~~
- *enter ~~two or more legal~~ a 10 and 25 coin*

Step 5: Refactor to remove duplication. Looking over the code I find no need for refactoring.

5.6 Iteration 4: Two Valid Coins

I decide to make the last item, *enter a 10 and 25 coin*, on the list my focus for the next iteration. This is because it will allow me both to complete the coin validation functionality as well as drive the summation of inserted payment into existence.

Step 1: Quickly add a test is trivial, I add a `shouldDisplay14MinFor10And25Cents` method with the proper input and expected values. *Step 2: Run all tests and see the new one fail* provides the answer:

Sidebar 5.3: Test Code Must Obey Production Code Invariants

One nasty experience that some of my students have run into is that their testing code is initially too simple because it relies on the immature production code, the stub code, that always is the initial starting point for a class in TDD. This is best illustrated by example.

Consider our first two iterations of the pay station: accepting a coin and calculating rates. Actually I could have made these iterations by using the value 15 as input parameter to addPayment, and for instance tested the rate calculation by:

```
@Test
public void reallyBadTest() throws IllegalCoinException {
  ps.addPayment( 15 );
  assertEquals( 15/5*2, ps.readDisplay() );
}
```

This would lead to two test cases passing and even to the correct implementation. However, later when I introduce validation of coins, the test case above would suddenly fail and I have to refactor the test case.

An even more subtle example is when the production code behavior is guided by its internal state. As an example, consider implementing a chess program. In chess the white player always moves first. However, if you start your iterations by testing valid and invalid movement for a pawn and happen to use a black pawn then your test cases will pass up until you start implementing the code that handles turn taking. Now your test cases fail because it is not black's turn to move.

The morale is that your testing code must always follow the invariants, preconditions, and specifications of the classes it tests: provide proper and valid parameters, call methods in the correct sequence, etc. Otherwise your test cases will probably break in later iterations.

```
JUnit version 4.4
....E
Time: 0,016
There was 1 failure:
1) shouldDisplay14MinFor10And25Cents(TestPayStation)
IllegalCoinException: Invalid coin: 10
  [...]
```

Step 3: Make a little change is also trivial, inserting the missing case in the switch (line 5) in addPayment.

```
1   public void addPayment( int coinValue )
2           throws IllegalCoinException {
3     switch ( coinValue ) {
4     case 5: break;
5     case 10: break;
6     case 25: break;
7     default:
8       throw new IllegalCoinException("Invalid coin: "+coinValue);
9     }
10    insertedSoFar = coinValue;
11  }
```

When I run the tests again, however, the test again fails but for another reason:

```
JUnit version 4.4
....E
Time: 0,016
There was 1 failure:
1) shouldDisplay14MinFor10And25Cents(TestPayStation)
java.lang.AssertionError: Should display 14 min for 10+25 cents
    expected:<14> but was:<10>
```

This is actually quite typical—that you add a test and then have several mini iterations of adding production code, test fails, modify production code, test still fails, modify, etc., until finally the test case pass. Here, the failed test drives the summation of payment into the production code, the missing "+=". Thus the final code of the iteration becomes:

Fragment: chapter/tdd/iteration-4/PayStationImpl.java

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
}
```

Step 4: Run all tests and see them all succeed. Four tests pass. Again no need for *Step 5: Refactor to remove duplication.*

✻ ~~accept legal coin~~
✻ ~~reject illegal coin, exception~~
✻ ~~5 cents should give 2 minutes parking time.~~
✻ ~~readDisplay~~
✻ buy produces valid receipt
✻ cancel resets pay station
✻ ~~25 cents = 10 minutes~~
✻ ~~enter a 10 and 25 coin~~

Done.

5.7 Iteration 5: Buying (Faked)

OK, now I have enough production in place that I find the item *buy produces valid receipt* is a suitable *One Step Test*. Completing this item also completes the *Buy a parking ticket* story defined in Section 4.1. To remind you, the buy method looks like this

Fragment: chapter/tdd/iteration-5/PayStation.java

```
/**
 * Buy parking time. Terminate the ongoing transaction and
 * return a parking receipt. A non-null object is always returned.
 * @return a valid parking receipt object.
 */
public Receipt buy();
```

The buy method returns an object defined by the interface Receipt. This interface just contains a single method, **int** value(), that returns the number of minutes parking time the receipt object represents.

Step 1: Quickly add a test. Why not try all the three types of coins:

Fragment: chapter/tdd/iteration-5/TestPayStation.java

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

I here use another of JUnit's assertions: assertNotNull that tests whether the argument object reference is not null. Thus I first test that we get a receipt object and second that the parking time it represents is correct.

Exercise 5.1: You may observe that if the buy method returns a null reference, then the second assertEquals will report failure. One may argue that the first assertNotNull is then redundant. Argue in favor of having both asserts in the testing code.

I go through *Step 2: Run all tests and see the new one fail.* As I have not made any implementation effort on the buy method, it simply returns null and the output from JUnit is therefore no surprise.

```
1) shouldReturnCorrectReceiptWhenBuy(TestPayStation)
java.lang.AssertionError: Receipt reference should not be null
```

Now, next is *Step 3: Make a little change* but this poses a problem. The buy operation requires two steps: implementing the buy method in PayStationImpl and writing an implementation of the Receipt interface. *Take small steps* races through my head! I need to break this complex problem into two smaller ones that can be done in isolation: getting buy to work and getting receipt to work. The question is: in which order? The buy operation depends upon the receipt implementation, not the other way around, so the most obvious step is to make the receipt work and then go back and get the buy operation in place. Agree?

☞ Take a moment to reflect upon the two ways to go before reading on: A) finish buy and then get receipt to work or B) get receipt to work first, and then go back to the buy operation. What are the benefits and liabilities of each approach?

The answer to this question may at first seem counter intuitive but the arguably best approach is to finish the iteration on the buy test first! You may argue that this is impossible but if you do, then you have not truly appreciated the *Fake It* principle. By using *Fake It* we can break the problem into smaller steps that each can be finished:

1. Keep the focus on the ongoing iteration to get a receipt that reads ((5+10+25) / 5 * 2) minutes by returning a *Fake It* receipt.
2. In the next iteration, implement a proper receipt class.
3. And finally, in an iteration use *Triangulation* to finish the buy operation.

I must remember this analysis by updating my test list to ensure that the fake receipt does not stay in the production code.

✻ ~~accept legal coin~~
✻ ~~reject illegal coin, exception~~
✻ ~~5 cents should give 2 minutes parking time.~~
✻ ~~readDisplay~~
✻ buy for 40 cents produces valid receipt
✻ cancel resets pay station
✻ ~~25 cents = 10 minutes~~
✻ ~~enter a 10 and 25 coin~~
✻ *receipt can store values*
✻ *buy for 100 cents*

The argument for sticking to the buy operation is *to keep focus* even if it means writing a bit more code in the short run. If you start getting Receipt in place at this moment then you break the focus on the buy iteration and introduce a new focus and therefore a new iteration. This behavior is unfortunately all too well known to seasoned developers: *"In order to fix defect A I must first get method B in place. Implementing method B I find that a new class C would be a good idea. Class C turned out to be quite a bit more complex than I thought and required class D to be refactored. Refactoring it exposed a defect in class E that I had to fix first. . . "* Before you know it you have spent a week implementing, debugging, testing, and find out that you have completely forgotten *why* all the effort was started in the first place. You have lost focus. And probably introduced code (and defects!) that do not do the original problem any good. It should be avoided.

Now I can finish the "buy for 40 cents" operation by using *Fake It*. An anonymous inner class comes in handy here. If anonymous classes are new to you, you can find a small description in sidebar 5.4.

Fragment: chapter/tdd/iteration-5/PayStationImpl.java

```java
public Receipt buy() {
  return new Receipt() {
      public int value() { return (5+10+25)/5*2; }
    };
}
```

Step 4: Run all tests and see them all succeed gives me the comforting signal that all tests pass. No *Step 5: Refactor to remove duplication* this time either.

Sidebar 5.4: Anonymous Classes in Java

Java allows *inner classes*, that is, a class that is defined inside another class.

```
class OuterClass {
    [...]
    class InnerClass {
        [...]
    }
}
```

The normal scoping rules apply so an instance of InnerClass only exists as part of an instance of the OuterClass. A special case is an *anonymous* class which is an inner class declared without naming it. Such an anonymous class is declared simply by defining the method bodies as part of the object creation using new. Thus

Fragment: chapter/tdd/iteration-5/PayStationImpl.java

```
    return new Receipt() {
        public int value() { return (5+10+25)/5*2; }
      };
```

means: *create a new instance of an anonymous class that implements* Receipt *and has this implementation of the* value *method.*

Anonymous inner classes are very handy when using *Fake It* to create a “constant”, first, implementation of an interface, as you do not have to create classes in their own Java source code files.

You can consult the *Java Tutorial* at Sun’s website for more details on inner classes.

5.8 Iteration 6: Receipt

Fake It should never stay in the production code for long. This requires me to address the *receipt can store values* item. The only method in the receipt is the value() method.

Step 1: Quickly add a test. The receipt must represent a parking time value, so let us express this in a test case. Writing the test case involves a bit of design. value() must return the number of minutes parking time it represents but how does the pay station “tell” the receipt this value? First things first:

TDD Principle: Assert First

When should you write the asserts? Try writing them first.

That is, get the assertions in place first and return to the problem of setting up the stuff to be asserted later. First shot:

```
  @Test
  public void shouldStoreTimeInReceipt() {
    ...
    assertEquals( "Receipt can store 30 minute value",
                  30, receipt.value() );
  }
```

Where did receipt come from? I have to create it (remember, I am testing Receipt only). How do I do that? I invoke new on the constructor. As receipts are objects whose state should not change after they have been made, it makes good sense to provide the constructor with the minute value directly. I complete the test case:

Fragment: chapter/tdd/iteration-6/TestPayStation.java

```
  @Test
  public void shouldStoreTimeInReceipt() {
    Receipt receipt = new ReceiptImpl(30);
    assertEquals( "Receipt can store 30 minute value",
                  30, receipt.value() );
  }
```

Step 2: Run all tests and see the new one fail is this time not indicated by a failed test: it is the compiler that complains loudly:

```
TestPayStation.java:81: cannot find symbol
symbol  : class ReceiptImpl
location: class TestPayStation
    Receipt receipt = new ReceiptImpl(30);
                          ^
1 error
```

Step 3: Make a little change means writing an implementation of the Receipt interface. I often use the convention to name implementation classes with an Impl appended to the interface name, so I create a java source file named ReceiptImpl. Next, I realize that the complexity of ReceiptImpl is very low: it is a matter of assigning a single instance variable in the constructor and return it in the value method. Testing should always be a question of cost versus benefit and with such a straight forward implementation I decide that *Fake It* and *Triangulation* are too small steps. When this is the case use:

TDD Principle: **Obvious Implementation**

How do you implement simple operations? Just implement them.

Consequently, I simply add the new class to the project and introduce all the necessary code in one go:

Listing: chapter/tdd/iteration-6/ReceiptImpl.java

```
/** Implementation of Receipt.
*/

public class ReceiptImpl implements Receipt {
  private int value;
  public ReceiptImpl(int value) { this.value = value; }
  public int value() { return value;}
}
```

Step 4: Run all tests and see them all succeed. The passed tests show we are on track. Again, the code is so small that there is no duplication, so I just update the test list.

- ~~accept legal coin~~
- ~~reject illegal coin, exception~~
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- ~~buy for 40 cents produces valid receipt~~
- cancel resets pay station
- ~~25 cents = 10 minutes~~
- ~~enter a 10 and 25 coin~~
- ~~receipt can store values~~
- buy for 100 cents

☞ Is one test case enough, only testing with value 30 minutes? Consider the *Representative Data* principle.

5.9 Iteration 7: Buying (Real)

The last iteration ended in a proper implementation of the receipt, so the obvious next choice is to triangulate the proper buy behavior, item *buy for 100 cents.*

Step 1: Quickly add a test:

Fragment: chapter/tdd/iteration-7/TestPayStation.java

```
@Test
public void shouldReturnReceiptWhenBuy100c()
  throws IllegalCoinException {
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(25);
  ps.addPayment(25);

  Receipt receipt;
  receipt = ps.buy();
  assertEquals((5*10+2*25) / 5 * 2 , receipt.value() );
}
```

which of course leads to *Step 2: Run all tests and see the new one fail*.

At this point you may be concerned with the long list of addPayment method calls. As a programmer, trained to spot opportunities for parameterizing code, you would want to "do something clever" here. Maybe generalize the coin input to make an array of coin values and then make a loop that iterates over the array and call addPayment on each element. Maybe refactor the testing code to make a private method that takes a coin array as parameter and implements the loop.

But there is a problem with this line of thought. Automated test cases are expressed as source code, and we can just as easily make mistakes in the testing code as in the production code. Thus we may ask ourselves the question: *"Why can tests lead to higher reliability when they are simply more code that may contain more defects?"* The answer is that testing only provides value if I keep the tests very simple, evident and easy to read.

TDD Principle: **Evident Tests**

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

The code complexity of a loop over an array is much higher than the complexity of the simple list of addPayment invocations. You must always consider the complexity in the test code versus the complexity of the code that it is testing. If the testing code is more complex then odds are that defects will be in the testing code. For instance, a simple accessor method like readDisplay is so simple that we will not write a special test case for it (it gets tested in most of the test cases anyway). Test cases should ideally only contain assignments, method calls, and assertions, and you should avoid loops, recursion, and other "complex" structures if possible. Making private (simple) helper methods that can be used from multiple test methods is also OK. For instance, if I later need a test case for entering two dollars, I would refactor the above test case: move the list of addPayment calls into a private method in the test class named e.g. insertOneDollar(), and call it from the shouldReturnReceiptWhenBuy100c method.

Back to the pay station and *Step 3: Make a little change*. I remove the *Fake It* code and introduce a plausible implementation

```
public Receipt buy() {
  return new ReceiptImpl(insertedSoFar * 2 / 5);
}
```

which leads to *Step 4: Run all tests and see them all succeed*.

Finally, *Step 5: Refactor to remove duplication*. Looking over the production code again I discover duplicated code namely:

```
public int readDisplay() {
  return insertedSoFar * 2 / 5;
}
public Receipt buy() {
  return new ReceiptImpl(insertedSoFar * 2 / 5);
}
```

☞ Sketch some plausible designs that remove this duplication.

Two choices that come to my mind are A) to introduce a calculateParkingTime method to be called from both readDisplay and buy or B) introduce a instance variable timeBought that is updated in addPayment to keep track of the amount of parking time bought so far, and used in both readDisplay and buy .

The point is that whatever course taken, I now have several test cases to ensure that my refactoring of the production code does not alter pay station behavior! If I happen to introduce a defect during the implementation then most likely one of my test cases would break and I would have clear indication of what test case and what line number are causing the problem.

I have no strong opinions as to which solution is the better. I choose to introduce an instance variable timeBought that is set in the addPayment method:

Listing: chapter/tdd/iteration-7/PayStationImpl.java

```
/** Implementation of the pay station.
*/

public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  public void addPayment( int coinValue )
          throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar += coinValue;
    timeBought = insertedSoFar / 5 * 2;
  }
  public int readDisplay() {
    return timeBought;
  }
  public Receipt buy() {
    return new ReceiptImpl(timeBought);
  }
  public void cancel() {
  }
}
```

and the passed tests tell me that my refactoring was OK: behavior has not changed even though I have made modifications in several methods.

Is this all there is to buying? I look over the original story **Buy a parking ticking** from Chapter 4 and find that there was one additional requirement: the pay station should be cleared and prepared for a new transaction. Presently, payments are simply accumulating from one driver using the station to the next. I put it on the test list:

- ~~accept legal coin~~
- ~~reject illegal coin, exception~~
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- ~~buy for 40 cents produces valid receipt~~
- cancel resets pay station
- ~~25 cents = 10 minutes~~
- ~~enter a 10 and 25 coin~~
- ~~receipt can store values~~
- ~~buy for 100 cents~~
- *clearing after a buy operation*

5.10 Iteration 8: Clearing after Buy

Step 1: Quickly add a test is done by testing that the display is cleared after a buy—and that doing a following buy scenario behaves as expected. I express this requirement as a test case:

Fragment: chapter/tdd/iteration-8/TestPayStation.java

```
@Test
public void shouldClearAfterBuy()
        throws IllegalCoinException {
  ps.addPayment(25);
  ps.buy(); // I do not care about the result
  // verify that the display reads 0
  assertEquals( "Display should have been cleared",
                0 , ps.readDisplay() );
  // verify that a following buy scenario behaves properly
  ps.addPayment(10); ps.addPayment(25);
  assertEquals( "Next add payment should display correct time",
                (10+25) / 5 * 2, ps.readDisplay() );
  Receipt r = ps.buy();
  assertEquals( "Next buy should return valid receipt",
                (10+25) / 5 * 2, r.value() );
  assertEquals( "Again, display should be cleared",
                0 , ps.readDisplay() );
}
```

Please note that *Isolated Test* does not mean that you have to make a test method for every assert you make. This test case seek to verify that the pay station is properly cleared after one buy scenario has been completed and this entails several aspects: that the display once again reads 0 minutes, that a second adding of payment will properly show number of minutes parking time in the display, and that the resulting receipt from the second buy will indeed show the proper value. Thus in this test case several asserts are required.

Step 2: Run all tests and see the new one fail reports failure as the present code accumulates the inserted amount even over multiple usages:

```
JUnit version 4.4
.......E
Time: 0,078
There was 1 failure:
1) shouldClearAfterBuy(TestPayStation)
java.lang.AssertionError: Display should have been cleared
   expected:<0> but was:<10>
   [lines omitted]
```

The *Step 3: Make a little change* is simple as clearing code can be localized to the buy method so I change it to:

Fragment: chapter/tdd/iteration-8/PayStationImpl.java

```
public Receipt buy() {
  Receipt r = new ReceiptImpl(timeBought);
  timeBought = insertedSoFar = 0;
  return r;
}
```

and I achieve *Step 4: Run all tests and see them all succeed*. There is no need for *Step 5: Refactor to remove duplication*.

5.11 Iteration 9: Cancelling

One item left only on the test list.

- ✻ ~~accept legal coin~~
- ✻ ~~reject illegal coin, exception~~
- ✻ ~~5 cents should give 2 minutes parking time.~~
- ✻ ~~readDisplay~~
- ✻ ~~buy for 40 cents produces valid receipt~~
- ✻ cancel resets pay station
- ✻ ~~25 cents = 10 minutes~~
- ✻ ~~enter a 10 and 25 coin~~
- ✻ ~~receipt can store values~~
- ✻ ~~buy for 100 cents~~
- ✻ ~~clearing after a buy operation~~

Step 1: Quickly add a test, the test case is similar in structure to the one for "clearing after buy", I insert a few coins, cancel, and verify that the display shows 0 minutes.

Fragment: chapter/tdd/iteration-9/TestPayStation.java

```
@Test
public void shouldClearAfterCancel()
        throws IllegalCoinException {
  ps.addPayment(10);
  ps.cancel();
  assertEquals( "Cancel should clear display",
                0 , ps.readDisplay() );
  ps.addPayment(25);
  assertEquals( "Insert after cancel should work",
                25/5*2 , ps.readDisplay() );
}
```

Note again that I am a bit precautious and also verify that adding coins after the cancel works as expected.

Step 2: Run all tests and see the new one fail, yep, quickly on to *Step 3: Make a little change* where I can simply duplicate the clearing code used in the buy:

```
public void cancel() {
  timeBought = insertedSoFar = 0;
}
```

This leads to *Step 4: Run all tests and see them all succeed*. And in *Step 5: Refactor to remove duplication* I obviously have duplicated code: the clearing code. I find that a private method whose responsibility it is to reset the pay station is the proper solution for this problem.

In conclusion, I have now through nine iterations implemented nine test cases that give me high confidence that they cover the requirements put forward by the municipality of Alphatown. Their requirements have been expressed as test cases that I have written first and these have in turn driven the implementation to its final state:

Listing: chapter/tdd/iteration-9/PayStationImpl.java

```
/** Implementation of the pay station.
*/
```

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  public void addPayment( int coinValue )
          throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar += coinValue;
    timeBought = insertedSoFar / 5 * 2;
  }
  public int readDisplay() {
    return timeBought;
  }
  public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    reset();
    return r;
  }
  public void cancel() {
    reset();
  }
  private void reset() {
    timeBought = insertedSoFar = 0;
  }
}
```

After this long chapter I think it is time to list a last but not least important principle.

TDD Principle: **Break**

What do you do when you feel tired or stuck? Take a break.

Programming is a highly demanding, creative, process. If you get too tired when programming then you introduce more defects than you remove and progress is slow or even negative. When you get stuck, then go for coffee, take some fresh air, or chat with your colleagues. Odds are that you have a fresh look and a better idea when you get back.

5.12 The Test-Driven Process

After this long journey in the details, it is time to look at the benefits and liabilities of using a test-driven development process. Some important advantages are:

- *Clean code that works* is a mantra of TDD. TDD aids in writing code that is maintainable (clean) and reliable (works). The reliability aspect is supported by the growing test suite that is run all the time and in which each test case verifies correct behavior of some feature of the production code. The maintainability aspect is supported by the refactoring step in the rhythm that ensure that after each iteration the code is cleaned for duplication or other less maintainable aspects.

- *Fast feedback gives programmer confidence.* The focus is to apply the rhythm's five steps in fast iterations, take small steps, and keep focus. All these contribute to breaking up problems into small, manageable, steps that are more quickly solved. This gives you more confidence as each test that passes is a success and one step closer to finishing your task. Another important advantage is that the location in the production code that is the source of a problem is known: it is almost always the source code added in the present iteration. Thus the time spent on locating a defect is reduced. Contrast this to development processes where there is a time span of weeks between running and testing the code base: you can produce a lot of (untested) code in a week and a defect may be anywhere.
- *Strong focus on reliable software.* A major benefit of writing automated tests is that they become valuable assets that give confidence that our software works as intended even when we make major refactorings. If test cases are not present most developers are simply too afraid to make any dramatic changes in large systems. An old saying is: *"If it ain't broke, don't fix it."* Software development becomes driven by fear of introducing defects. A large suite of high quality tests counters this fear and allows us to quickly assess if a dramatic change will work or not. If 57 test cases of 50,000 break after introducing the change then I am pretty confident that the change is worthwhile to introduce; if 49,000 out of 50,000 break I will think twice before proceeding...
- *Playing with the interface from the client's side.* TDD forces us to write the test first ergo I must write the code that *calls* methods on an object *before* I write the implementation. Therefore you look at the object from the client's point of view (the user of the object) instead of from the server's point of view (the object itself). This counters the force often seen in practice that developers implement all sorts of wonderful behavior that is never used and/or methods that have odd names with weird parameter lists.
- *Documentation of interface and protocol.* Test cases are reliable documentation of a class or a large software unit. They teach you how a class works: what methods it has, the parameters of the methods, and the order in which methods should be invoked, etc. It does not replace higher-level documentation, but maintained test cases are documentation that is not out-of-date and show a class' usage at the most detailed level.
- *No "driver" code.* Early in a project's development life cycle there is often no graphical user interface or any interface at all. Therefore in traditional development small "drivers" are often written: small programs that set up a few objects, call a few methods, and write a bit of output in a shell. Such drivers are seldom maintained and are quickly outdated. In TDD these short-lived programs are replaced by the test cases which in contrast are maintained and keep being an important asset. Thus TDD trades time making driver programs that are of no value in later stages for time making test cases that keep their value through-out the project's life time.
- *Structured programming process.* The set of testing principles: *Fake It*, *Triangulation*, etc.; provides me with a range of possible options to take in the program-

> ming process. I do not have a completely "blank paper" where only the problem statement is given but no clues as how to get the result, instead I have a lot of options. In my own 20 year professional life with programming I have developed intuition regarding "what to do" but it has mostly been "gut-feeling" and therefore communicating what I do now and what I will do next has been difficult. In contrast, the TDD principles form a set of options that I can name, consider, discuss, choose, and apply.

No rose without thorns, however. One issue that often pops up in TDD is that if an interface needs to be changed for some reason then it comes with a penalty as all the test cases that use this interface have to be rewritten as well. This can be quite costly especially if the change is not just syntactically but the logic is changed. Many are tempted simply to dump the test cases—but if you do so then you have also lost your means to ensure that no defects are introduced if you need to refactor. TDD is very strict about this: test cases *must* be maintained. Another issue is the quality of the test cases. They, too, must be refactored and kept small and easily readable to keep their value over time.

The term *driven* in test-driven development is central. The tests *drive* the production code. In its extreme interpretation you must *never* implement any production code that is not driven from the tests. If you find yourself making production code that does more than is warranted by the test cases—then don't! Instead make a note on the test list of a test that *will* require this code and do another iteration.

5.13 Summary of Key Concepts

Test-driven development is a program development process that focuses on writing reliable and maintainable software: *Clean code that works.* TDD is based on the values of *simplicity*, *keep focus*, and *take small steps.* Simplicity is about keeping things simple and avoiding implementing code that is really not needed, while keeping focus and taking small steps are about concentrating on one implementation task at a time and making each small addition work before proceeding.

The TDD process is structured by the five step *rhythm* that is reproduced in the front inner cover of the book. The rhythm defines an iteration on the software in which one very small feature is added. First you add a new test case to your test suite, a test case that asserts the required or specified behavior of the feature. Next you implement production code until your new test case passes. Finally, you review your code to spot opportunities for refactoring, that is, improving the internal structure of the code to make it more maintainable while keeping the external behavior intact.

During each iteration the *TDD principles* express actions to take, or rules to follow in order to keep your testing and production code reliable and maintainable. The principles are also reproduced on the inner cover.

The benefits of TDD is confidence in your programming effort as you can see all the tests passing, a high focus on reliability and maintainability of your code, and that the rhythm and the principles help you structure your programming process.

The test-driven development technique was invented by Kent Beck and Ward Cunningham. Kent Beck has written a number of excellent books on the topic as well as the larger frame it was invented within: Extreme Programming. The present chapter is highly influenced by the book *Test-Driven Development–By Example* (Beck 2003). Most of the testing principles outlined in this chapter are reprinted from Beck's book with permission.

5.14 Selected Solutions

Discussion of Exercise 5.1:

Tests are written for the programmer and therefore the more information you can get as programmer when a test case fails the better. In this case two things can go wrong: the receipt reference is null or it is valid but the minute contents of the valid Receipt object is wrong. I have added an assert to catch each of these situations to provide as precise information as possible. Otherwise I might try to find a wrong assignment of the minute value while what I should be looking for was a missing object creation.

5.15 Review Questions

Explain the values that TDD is based upon and why.

Describe the steps in the TDD rhythm. Explain the essence and motivation for the following TDD principles: Test, Test First, Test List, One Step Test, Fake It, Triangulation, Obvious Implementation, Isolated Test, Representative Data, and Evident Test. Mention some examples from the pay station where these principles are used.

Explain what refactoring is. How does TDD support you when you refactor code?

5.16 Further Exercises

Exercise 5.2. Source code directory:
`chapter/tdd/iteration-0`

Develop the pay station yourself using the TDD process.

1. Develop it using an alternative path i.e. pick the items from the test list in another order than I have done.
2. Add a method `int empty()` that returns the total amount of money earned by the pay station since last call; and "empties" it, setting the earning to zero. Implement it using TDD.

Exercise 5.3:

To simulate that you get your coins back when pushing the cancel button, the interface of the `cancel` method is changed into:

```
/** Cancel the present transaction. Resets the pay station for a
    new transaction.
    @return A Dictionary defining the coins returned to the user.
    The key is a valid coin type and the associated value is
    the number of these coins that are returned (i.e. identical
    to the number of this coin type that the user has entered.)
    The dictionary object is never null even if no coins
    have been entered.
*/
public Map<Integer,Integer> cancel();
```

1. Implement the cancel method using TDD.
2. Document your TDD process in a log book outlining the rhythm, TDD principles used, and final test list.
3. Describe your experience with TDD by outlining benefits and problems encountered. Reflect on your ability to *keep focus* and *take small steps.*

Exercise 5.4. Source code directory:
`exercise/tdd/breakthrough`

Breakthrough is a two person game played on a 8x8 chessboard, designed by Dan Troyka. Each player has 16 pieces, initially positioned on the two rows nearest to the player, as outlined in Figure 5.1. The winner is the first player to move one of his pieces to the *home row*, that is, the row farthest from the player. Players alternate to take turns, moving one piece per turn. White player begins.

The roles of movement is simple. A piece may move one square straight or diagonally forward towards the home row if that square is empty. A piece, however, may only capture an opponent piece diagonally. When capturing, the opponent piece is removed from the board and the player's piece takes its position, as you do in chess.

1. Develop an implementation of the Breakthrough interface using TDD.
2. Document your TDD process in a log book outlining the rhythm, TDD principles used, and final test list.
3. Describe your experience with TDD by outlining benefits and problems encountered. Reflect on your ability to *keep focus* and *take small steps.*

Exercise 5.5:

Exercise 2.3 and 2.4 defined a function to verify if an email adress is well-formed.

1. Implement the isValid method using TDD.
2. Document your TDD process in a log book outlining the rhythm, TDD principles used, and final test list.

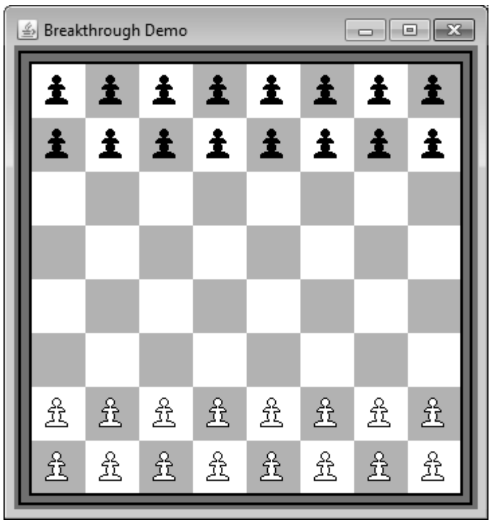

Figure 5.1: Initial position of Breakthrough

3. Describe your experience with TDD by outlining benefits and problems encountered. Reflect on your ability to *keep focus* and *take small steps.*

Note: Even if you use the regular expression library you should adhere to the test-*driven* paradigm: do *not* try to figure out the full regex and then add tests afterwards. Instead, take a simple *one step test* email address, adapt the regex minimally to make it pass, take a more complex email address, make the smallest change to the regex, etc.