8 Test-driven development

This chapter covers

- Understanding test-driven development
- Being productive with TDD
- When not to use TDD

Software developers are pretty used to the traditional development process. First, they implement. Then, and only then, they test. But why not do it the other way around? In other words, why not write a test first and then implement the production code?

In this chapter, we discuss this well-known approach: *test-driven development* (TDD). In a nutshell, TDD challenges our traditional way of coding, which has always been "write some code and then test it." With TDD, we start by writing a test representing the next small feature we want to implement. This test naturally fails, as the feature has not yet been implemented! We then make the test pass by writing some code. With the test now green, and knowing that the feature has been implemented, we go back to the code we wrote and refactor it.

TDD is a popular practice, especially among Agile practitioners. Before I dive into the advantages of TDD and pragmatic questions about working this way, let's look at a small example.

8.1 Our first TDD session

For this example, we will create a program that converts Roman numerals to integers. Roman numerals represent numbers with seven symbols:

- I, *unus*, 1, (one)
- V, *quinque*, 5 (five)
- X, *decem*, 10 (ten)
- L, *quinquaginta*, 50 (fifty)
- C, *centum*, 100 (one hundred)
- D, *quingenti*, 500 (five hundred)
- M, *mille*, 1,000 (one thousand)

To represent all possible numbers, the Romans combined the symbols, following these two rules:

- Digits of lower or equal value on the right are added to the higher-value digit.
- Digits of lower value on the left are subtracted from the higher-value digit.

For instance, the number XV represents 15 (10 + 5), and the number XXIV represents 24 (10 + 10 – 1 + 5).

The goal of our TDD session is to implement the following requirement:

> Implement a program that receives a Roman numeral (as a string) and returns its representation in the Arabic numeral system (as an integer).

Coming up with examples is part of TDD. So, think about different inputs you can give the program, and their expected outputs. For example, if we input `"I"` to the program, we expect it to return 1. If we input `"XII"` to the program, we expect it to return 12. Here are the cases I can think of:

- Simple cases, such as numbers with single characters:
  - If we input `"I"`, the program must return 1.
  - If we input `"V"`, the program must return 5.
  - If we input `"X"`, the program must return 10.
- Numbers composed of more than one character (without using the subtractive notation):
  - If we input `"II"`, the program must return 2.
  - If we input `"III"`, the program must return 3.
  - If we input `"VI"`, the program must return 6.
  - If we input `"XVII"`, the program must return 17.
- Numbers that use simple subtractive notation:
  - If we input `"IV"`, the program must return 4.
  - If we input `"IX"`, the program must return 9.

- Numbers that are composed of many characters and use subtractive notation:
  - If we input `"XIV"`, the program must return 14.
  - If we input `"XXIX"`, the program must return 29.

NOTE You may wonder about corner cases: What about an empty string? or null? Those cases are worth testing. However, when doing TDD, I first focus on the happy path and the business rules; I consider corners and boundaries later.

Remember, we are not in testing mode. We are in development mode, coming up with inputs and outputs (or test cases) that will guide us through the implementation. In the development flow I introduced in figure 1.4, TDD is part of "testing to guide development." When we are finished with the implementation, we can dive into rigorous testing using all the techniques we have discussed.

Now that we have a (short) list of examples, we can write some code. Let's do it this way:

1. Select the simplest example from our list of examples.
2. Write an automated test case that exercises the program with the given input and asserts its expected output. The code may not even compile at this point. And if it does, the test will fail, as the functionality is not implemented.
3. Write as much production code as needed to make that test pass.
4. Stop and reflect on what we have done so far. We may improve the production code. We may improve the test code. We may add more examples to our list.
5. Repeat the cycle.

The first iteration of the cycle focuses on ensuring that if we give `"I"` as input, the output is 1. The `RomanNumeralConverterTest` class contains our first test case.

Listing 8.1 Our first test method

```
public class RomanNumberConverterTest {
  @Test
  void shouldUnderstandSymbolI() {
    RomanNumeralConverter roman = new RomanNumeralConverter();
    int number = roman.convert("I");
    assertThat(number).isEqualTo(1);
  }
}
```

We will get compilation errors here, as the RomanNumeralConverter class does not exist!

At this moment, the test code does not compile, because the `RomanNumberConverter` class and its `convert()` method do not exist. To solve the compilation error, let's create some skeleton code with no real implementation.

Listing 8.2 Skeleton implementation of the class

```
public class RomanNumeralConverter {
  public int convert(String numberInRoman) {
```

```
    return 0;
  }
}
```

We do not want to return 0, but this makes the test code compile.

The test code now compiles. When we run it, it fails: the test expected 1, but the program returned 0. This is not a problem, as we expected it to fail. Steps 1 and 2 of our cycle are finished. It is now time to write as much code as needed to make the test pass—and it looks weird.

Listing 8.3 Making the test pass

```
public class RomanNumeralConverter {
  public int convert(String numberInRoman) {
    return 1;
  }
}
```

Returning 1 makes the test pass. But is this the implementation we want?

The test passes, but it only works in a single case. Again, this is not a problem: we are still working on the implementation. We are taking baby steps.

Let's move on to the next iteration of the cycle. The next-simplest example in the list is "If we input `"V"`, the program must return 5." Let's again begin with the test.

Listing 8.4 Our second test

```
@Test
void shouldUnderstandSymbolV() {
  RomanNumeralConverter roman = new RomanNumeralConverter();
  int number = roman.convert("V");
  assertThat(number).isEqualTo(5);
}
```

The new test code compiles and fails. Now let's make it pass. In the implementation, we can, for example, make the method `convert()` verify the content of the number to be converted. If the value is `"I"`, the method returns 1. If the value is `"V"`, it returns 5.

Listing 8.5 Making the tests pass

```
public int convert(String numberInRoman) {
  if(numberInRoman.equals("I")) return 1;
  if(numberInRoman.equals("V")) return 5;
  return 0;
}
```

Hard-coded ifs are the easiest way to make both tests pass.

The two tests pass. We could repeat the cycle for X, L, C, M, and so on, but we already have a good idea about the first thing our implementation needs to generalize: when the Roman numeral has only one symbol, return the integer associated with it. How can we implement such an algorithm?

- Write a set of `if` statements. The number of characters we need to handle is not excessive.

- Write a `switch` statement, similar to the `if` implementation.
- Use a map that is initialized with all the symbols and their respective integer values.

The choice is a matter of taste. I will use the third option because I like it best.

Listing 8.6 `RomanNumeralConverter` for single-character numbers

```
public class RomanNumeralConverter {

  private static Map<String, Integer> table =
      new HashMap<>() {{
        put("I", 1);
        put("V", 5);
        put("X", 10);
        put("L", 50);
        put("C", 100);
        put("D", 500);
        put("M", 1000);
      }};

  public int convert(String numberInRoman) {
    return table.get(numberInRoman);
  }
}
```

Declares a conversion table that contains the Roman numerals and their corresponding decimal numbers

Gets the number from the table

How do we make sure our implementation works? We have our (two) tests, and we can run them. Both pass. The production code is already general enough to work for single-character Roman numbers. But our test code is not: we have a test called `shouldUnderstandSymbolI` and another called `shouldUnderstandSymbolV`. This specific case would be better represented in a parameterized test called `shouldUnderstandOneCharNumbers`.

Listing 8.7 Generalizing our first test

```
public class RomanNumeralConverterTest {

  @ParameterizedTest
  @CsvSource({"I,1","V,5", "X,10","L,50",
  "C, 100", "D, 500", "M, 1000"})
  void shouldUnderstandOneCharNumbers(String romanNumeral,
    ➥ int expectedNumberAfterConversion) {
    RomanNumeralConverter roman = new RomanNumeralConverter();
    int convertedNumber = roman.convert(romanNumeral);
    assertThat(convertedNumber).isEqualTo(expectedNumberAfterConversion);
  }
}
```

Passes the inputs as a comma-separated value to the parameterized test. JUnit then runs the test method for each of the inputs.

NOTE The test code now has some duplicate code compared to the production code. The test inputs and outputs somewhat match the map in the production code. This is a common phenomenon when doing testing by example, and it will be mitigated when we write more complex tests.

We are finished with the first set of examples. Let's consider the second scenario: two or more characters in a row, such as II or XX. Again, we start with the test code.

Listing 8.8 Tests for Roman numerals with multiple characters

```
@Test
void shouldUnderstandMultipleCharNumbers() {
  RomanNumeralConverter roman = new RomanNumeralConverter();
  int convertedNumber = roman.convert("II");
  assertThat(convertedNumber).isEqualTo(2);
}
```

To make the test pass in a simple way, we could add the string "II" to the map.

Listing 8.9 Map with all the Roman numerals

```
private static Map<String, Integer> table =
  new HashMap<>() {{
    put("I", 1);

    put("II", 2);

    put("V", 5);
    put("X", 10);
    put("L", 50);
    put("C", 100);
    put("D", 500);
    put("M", 1000);
  }};
```

Adds II. But that means we would need to add III, IV, and so on—not a good idea!

If we did this, the test would succeed. But it does not seem like a good idea for implementation, because we would have to include all possible symbols in the map. It is time to generalize our implementation again. The first idea that comes to mind is to iterate over each symbol in the Roman numeral we're converting, accumulate the value, and return the total value. A simple loop should do.

Listing 8.10 Looping through each character in the Roman numeral

```
public class RomanNumeralConverter {
  private static Map<Character, Integer> table =
      new HashMap<>() {{
        put('I', 1);
        put('V', 5);
        put('X', 10);
        put('L', 50);
        put('C', 100);
        put('D', 500);
        put('M', 1000);
      }};

  public int convert(String numberInRoman) {
    int finalNumber = 0;
```

The conversion table only contains the unique Roman numeral.

Variable that aggregates the value of each Roman numeral

```
    for(int i = 0; i < numberInRoman.length(); i++) {
      finalNumber += table.get(numberInRoman.charAt(i));
    }

    return finalNumber;
  }
}
```

Gets each character's corresponding decimal value and adds it to the total sum

Note that the type of key in the map has changed from `String` to `Character`. The algorithm iterates over each character of the string `numberInRoman` using the `charAt()` method, which returns a `char` type. We could convert the `char` to `String`, but doing so would add an extra, unnecessary step.

The three tests continue passing. It is important to realize that our focus in this cycle was to make our algorithm work with Roman numerals with more than one character—we were not thinking about the previous examples. This is one of the main advantages of having the tests: we are sure that each step we take is sound. Any bugs we introduce to previously working behavior (*regression bugs*) will be captured by our tests.

We can now generalize the test code and exercise other examples that are similar to the previous one. We again use parameterized tests, as they work well for the purpose.

Listing 8.11 Parameterizing the test for multiple characters

```
@ParameterizedTest
@CsvSource({"II,2","III,3", "VI, 6", "XVIII, 18",
"XXIII, 23", "DCCLXVI, 766"})
void shouldUnderstandMultipleCharNumbers(String romanNumeral,
  ➥ int expectedNumberAfterConversion) {
  RomanNumeralConverter roman = new RomanNumeralConverter();
  int convertedNumber = roman.convert(romanNumeral);
  assertThat(convertedNumber).isEqualTo(expectedNumberAfterConversion);
}
```

Tries many inputs with multiple characters. Again, CSVSource is the simplest way to do this.

NOTE I chose the examples I passed to the test at random. Our goal is not to be fully systematic when coming up with examples but only to use the test cases as a safety net for developing the feature. When we're finished, we will perform systematic testing.

A single test method or many?

This test method is very similar to the previous one. We could combine them into a single test method, as shown here:

```
@ParameterizedTest
@CsvSource({
  // single character numbers
  "I,1","V,5", "X,10","L,50", "C, 100", "D, 500", "M, 1000",
  // multiple character numbers
  "II,2","III,3", "V,5","VI, 6", "XVIII, 18", "XXIII, 23", "DCCLXVI, 766"
})
```

All the inputs from the different tests are combined into a single method.

```
void convertRomanNumerals(String romanNumeral,
  ➥ int expectedNumberAfterConversion) {
  RomanNumeralConverter roman = new RomanNumeralConverter();
  int convertedNumber = roman.convert(romanNumeral);
  assertThat(convertedNumber).isEqualTo(expectedNumberAfterConversion);
}
```

This is a matter of taste and your team's preference. For the remainder of the chapter, I keep the test methods separated.

Our next step is to make the subtractive notation work: for example, `IV` should return 4. As always, let's start with the test. This time, we add multiple examples: we understand the problem and can take a bigger step. If things go wrong, we can take a step back.

Listing 8.12 Test for the subtractive notation

```
@ParameterizedTest
@CsvSource({"IV,4","XIV,14", "XL, 40",
"XLI,41", "CCXCIV, 294"})                    Provides many inputs that exercise
                                             the subtractive notation rule
void shouldUnderstandSubtractiveNotation(String romanNumeral,
  ➥ int expectedNumberAfterConversion) {
  RomanNumeralConverter roman = new RomanNumeralConverter();
  int convertedNumber = roman.convert(romanNumeral);
  assertThat(convertedNumber).isEqualTo(expectedNumberAfterConversion);
}
```

Implementing this part of the algorithm requires more thought. The characters in a Roman numeral, from right to left, increase in terms of value. However, when a numeral is smaller than its neighbor on the right, it must be subtracted from instead of added to the accumulator. Listing 8.13 uses a trick to accomplish this: the `multiplier` variable becomes -1 if the current numeral (that is, the current character we are looking at) is smaller than the last neighbor (that is, the character we looked at previously). We then multiply the current digit by `multiplier` to make the digit negative.

Listing 8.13 Implementation for the subtractive notation

```
public int convert(String numberInRoman) {
  int finalNumber = 0;
  int lastNeighbor = 0;          Keeps the last visited digit

  for(int i = numberInRoman.length() - 1; i >= 0; i--) {
                                 Loops through the characters, but now from right to left
   int current = table.get(numberInRoman.charAt(i));
                                 Gets the decimal value of the current Roman digit

   int multiplier = 1;
   if(current < lastNeighbor) multiplier = -1;
                                 If the previous digit was higher than the current one, multiplies the current digit by -1 to make it negative
```

```
    finalNumber +=
      table.get(numberInRoman.charAt(i)) * multiplier;

    lastNeighbor = current;
  }

  return finalNumber;
}
```

Adds the current digit to the finalNumber variable. The current digit is positive or negative depending on whether we should add or subtract it, respectively.

Updates lastNeighbor to be the current digit

The tests pass. Is there anything we want to improve in the production code? We use `numberInRoman.charAt(i)` when summing the final number, but this value is already stored in the `current` variable, so we can reuse it. Also, extracting a variable to store the current digit after it is multiplied by 1 or -1 will help developers understand the algorithm. We can refactor the code, as shown in the following listing, and run the tests again.

Listing 8.14 Refactored version

```
public int convert(String numberInRoman) {
  int finalNumber = 0;
  int lastNeighbor = 0;

  for(int i = numberInRoman.length() - 1; i >= 0; i--) {

    int current = table.get(numberInRoman.charAt(i));

    int multiplier = 1;
    if(current < lastNeighbor) multiplier = -1;

    int currentNumeralToBeAdded = current * multiplier;
    finalNumber += currentNumeralToBeAdded;

    lastNeighbor = current;
  }

  return finalNumber;
}
```

Keeps the last digit visited

Uses the current variable and introduces the currentNumeralToBeAdded variable

Now that we have implemented all the examples in our initial list, we can think of other cases to handle. We are not handling invalid numbers, for example. The program must reject inputs such as `"VXL"` and `"ILV"`. When we have new examples, we repeat the entire procedure until the whole program is implemented. I will leave that as an exercise for you—we have done enough that we are ready to more formally discuss TDD.

8.2 Reflecting on our first TDD experience

Abstractly, the cycle we repeated in the previous section's development process was as follows:

1. We wrote a (unit) test for the next piece of functionality we wanted to implement. The test failed.

2 We implemented the functionality. The test passed.
3 We refactored our production and test code.

This TDD process is also called the *red-green-refactor cycle*. Figure 8.1 shows a popular way to represent the TDD cycle.

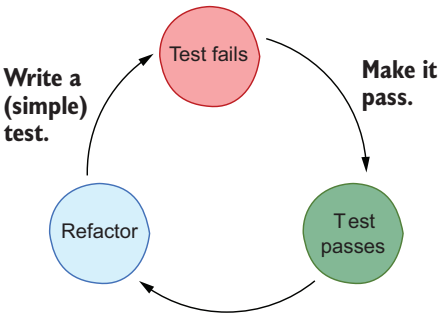


Figure 8.1 TDD, also known as the red-green-refactor cycle

TDD practitioners say this approach can be very advantageous for the development process. Here are some of the advantages:

- *Looking at the requirements first*—In the TDD cycle, the tests we write to support development are basically executable requirements. Whenever we write one of them, we reflect on what the program should and should not do.

  This approach makes us write code for the specific problem we are supposed to solve, preventing us from writing unnecessary code. And exploring the requirement systematically forces us to think deeply about it. Developers often go back to the requirements engineer and ask questions about cases that are not explicit in the requirement.
- *Full control over the pace of writing production code*—If we are confident about the problem, we can take a big step and create a test that involves more complicated cases. However, if we are still unsure how to tackle the problem, we can break it into smaller parts and create tests for these simpler pieces first.
- *Quick feedback*—Developers who do not work in TDD cycles produce large chunks of production code before getting any feedback. In a TDD cycle, developers are forced to take one step at a time. We write one test, make it pass, and reflect on it. These many moments of reflection make it easier to identify new problems as they arise, because we have only written a small amount of code since the last time everything was under control.
- *Testable code*—Creating the tests first makes us think from the beginning about a way to (easily) test the production code before implementing it. In the traditional flow, developers often think about testing only in the later stages of developing a feature. At that point, it may be expensive to change how the code works to facilitate testing.

- *Feedback about design*—The test code is often the first client of the class or component we are developing. A test method instantiates the class under test, invokes a method passing all its required parameters, and asserts that the method produces the expected results. If this is hard to do, perhaps there is a better way to design the class. When doing TDD, these problems arise earlier in the development of the feature. And the earlier we observe such issues, the cheaper it is to fix them.

NOTE TDD shows its advantages best in more complicated problems. I suggest watching James Shore's YouTube playlist on TDD (2014), where he TDDs an entire software system. I also recommend Freeman and Pryce's book *Growing Object-Oriented Systems Guided by Tests* (2009). They also TDD an entire system, and they discuss in depth how they use tests to guide their design decisions.

8.3 TDD in the real world

This section discusses the most common questions and discussions around TDD. Some developers love TDD and defend its use fiercely; others recommend not using it.

As always, software engineering practices are not silver bullets. The reflections I share in this section are personal and not based on scientific evidence. The best way to see if TDD is beneficial for you is to try it!

8.3.1 To TDD or not to TDD?

Skeptical readers may be thinking, "I can get the same benefits without doing TDD. I can think more about my requirements, force myself to only implement what is needed, and consider the testability of my class from the beginning. I do not need to write tests for that!" That is true. But I appreciate TDD because it gives me a rhythm to follow. Finding the next-simplest feature, writing a test for it, implementing nothing more than what is needed, and reflecting on what I did gives me a pace that I can fully control. TDD helps me avoid infinite loops of confusion and frustration.

The more defined development cycle also reminds me to review my code often. The TDD cycle offers a natural moment to reflect: as soon as the test passes. When all my tests are green, I consider whether there is anything to improve in the current code.

Designing classes is one of the most challenging tasks of a software engineer. I appreciate the TDD cycle because it forces me to use the code I am developing from the very beginning. The perception I have about the class I am designing is often different from my perception when I try to use the class. I can combine both of these perceptions and make the best decision about how to model the class.

If you write the tests after the code, and not before, as in TDD, the challenge is making sure the time between writing code and testing is small enough to provide developers with timely feedback. Don't write code for an entire day and then start testing—that may be too late.

8.3.2 TDD 100% of the time?

Should we always use TDD? My answer is a *pragmatic* "no." I do a lot of TDD, but I do not use TDD 100% of the time. It depends on how much I need to learn about the feature I am implementing:

- I use TDD when I don't have a clear idea of how to design, architect, or implement a specific requirement. In such cases, I like to go slowly and use my tests to experiment with different possibilities. If I am working on a problem I know well, and I already know the best way to solve the problem, I do not mind skipping a few cycles.
- I use TDD when dealing with a complex problem or a problem I lack the expertise to solve. Whenever I face a challenging implementation, TDD helps me take a step back and learn about the requirements as I go by writing very small tests.
- I do *not* use TDD when there is nothing to be learned in the process. If I already know the problem and how to best solve it, I am comfortable coding the solution directly. (Even if I do not use TDD, I always write tests promptly. I never leave it until the end of the day or the end of the sprint. I code the production code, and then I code the test code. And if I have trouble, I take a step back and slow down.)

TDD creates opportunities for me to learn more about the code I am writing from an implementation point of view (does it do what it needs to do?) as well as from a design point of view (is it structured in a way that I want?). But for some complex features, it's difficult even to determine what the first test should look like; in those cases, I do not use TDD.

We need ways to stop and think about what we are doing. TDD is a perfect approach for that purpose, but not the only one. Deciding when to use TDD comes with experience. You will quickly learn what works best for you.

8.3.3 Does TDD work for all types of applications and domains?

TDD works for most types of applications and domains. There are even books about using it for embedded systems, where things are naturally more challenging, such as Grenning's book *Test Driven Development for Embedded C* (2011). If you can write automated tests for your application, you can do TDD.

8.3.4 What does the research say about TDD?

TDD is such a significant part of software development that it is no wonder researchers try to assess its effectiveness using scientific methods. Because so many people treat it as a silver bullet, I strongly believe that you should know what practitioners think, what I think, and what research currently knows about the subject.

Research has shown several situations in which TDD can improve class design:

- Janzen (2005) showed that TDD practitioners, compared to non-TDDers, produced less-complex algorithms and test suites that covered more.
- Janzen and Saiedian (2006) showed that the code produced using TDD made better use of object-oriented concepts, and responsibilities were better distributed into different classes. In contrast, other teams produced more procedural code.
- George and Williams (2003) showed that although TDD can initially reduce the productivity of inexperienced developers, 92% of the developers in a qualitative analysis thought that TDD helped improve code quality.
- Dogša and Batič (2011) also found an improvement in class design when using TDD. According to the authors, the improvement resulted from the simplicity TDD adds to the process.
- Erdogmus et al. (2005) used an experiment with 24 undergraduate students to show that TDD increased their productivity but did not change the quality of the produced code.
- Nagappan and colleagues (2008) performed three case studies at Microsoft and showed that the pre-release defect density of projects that were TDD'd decreased 40 to 90% in comparison to projects that did not do TDD.

Fucci et al. (2016) argue that the important aspect is writing tests (before or after). Gerosa and I (2015) have made similar observations after interviewing many TDD practitioners. This is also the perception of practitioners. To quote Michael Feathers (2008), "That's the magic, and it's why unit testing works also. When you write unit tests, TDD-style or after your development, you scrutinize, you think, and often you prevent problems without even encountering a test failure."

However, other academic studies show inconclusive results for TDD:

- Müeller and Hagner (2002), after an experiment with 19 students taking a one-semester graduate course on extreme programming, observed that test-first did not accelerate implementation compared to traditional approaches. The code written with TDD was also not more reliable.
- Siniaalto and Abrahamsson (2007) compared five small-scale software projects using different code metrics and showed that the benefits of TDD were not clear.
- Shull and colleagues (2010), after summarizing the findings of 14 papers on TDD, concluded that TDD shows no consistent effect on internal code quality. This paper is easy to read, and I recommend that you look at it.

As an academic who has read most of the work on this topic, I find that many of these studies—both those that show positive effects and those that do not—are not perfect. Some use students, who are not experts in software development or TDD. Others use toy projects without specific room for TDD to demonstrate its benefits. And some use code metrics such as coupling and cohesion that only partially measure code quality. Of course, designing experiments to measure the benefits of a software engineering

practice is challenging, and the academic community is still trying to find the best way to do it.

More recent papers explore the idea that TDD's effects may be due not to the "write the tests first" aspect but rather to taking baby steps toward the final goal. Fucci et al. (2016) argue that "the claimed benefits of TDD may not be due to its distinctive test-first dynamic, but rather due to the fact that TDD-like processes encourage fine-grained, steady steps that improve focus and flow."

I suggest that you give TDD a chance. See if it fits your way of working and your programming style. You may decide to adopt it full-time (like many of my colleagues) or only in a few situations (like me), or you may choose never to do it (also like many of my colleagues). It is up to you.

8.3.5 Other schools of TDD

TDD does not tell you how to start or what tests to write. This flexibility gave rise to various different schools of TDD. If you are familiar with TDD, you may have heard of the London school of TDD, mockist vs. classicist TDD, and outside-in TDD. This section summarizes their differences and points you to other material if you want to learn more.

In the *classicist school of TDD* (or the *Detroit school of TDD*, or *inside-out TDD*), developers start their TDD cycles with the different units that will compose the overall feature. More often than not, classicist TDDers begin with the entities that hold the main business rules; they slowly work toward the outside of the feature and connect these entities to, say, controllers, UIs, and web services. In other words, classicists go from the inside (entities and business rules) to the outside (interface with the user).

Classicists also avoid mocks as much as possible. For example, when implementing a business rule that would require the interaction of two or more other classes, classicists would focus on testing the entire behavior at once (all the classes working together) without mocking dependencies or making sure to test the units in a fully isolated manner. Classicists argue that mocks reduce the effectiveness of the test suite and make test suites more fragile. This is the same negative argument we discussed in chapter 6.

The *London school of TDD* (or *outside-in TDD*, or *mockist TDD*), on the other hand, prefers to start from the outside (such as the UI or the controller that handles the web service) and then slowly work toward the units that will handle the functionality. To do so, they focus on how the different objects will collaborate. And for that to happen in an outside-in manner, these developers use mocks to explore how the collaboration will work. They favor testing isolated units.

Both schools of thought use the test code to learn more about the design of the code being developed. I like the way Test Double (2018) puts it: "In [the] Detroit school, if an object is hard to test, then it's hard to use; in [the] London school, if a dependency is hard to mock, then it's hard to use for the object that'll be using it."

My style is a mixture of both schools. I start from the inside, coding entities and business rules, and then slowly work to the outside, making the external layers call

these entities. However, I favor unit testing as much as possible: I do not like the tests of unit A breaking due to a bug in unit B. I use mocks for that, and I follow all the practices discussed in chapter 6.

I suggest that you learn more about both schools. Both have good points, and combining them makes sense. I recommend Mancuso's 2018 talk, which elaborates on the differences between the schools and how the approaches can be used.

8.3.6 TDD and proper testing

Some studies show that TDD practitioners write more test cases than non-TDD practitioners. However, I do not believe that the test suites generated by TDD sessions are as good as the strong, systematic test suites we engineered in the previous chapters after applying different testing practices. The reasoning is simple: when doing TDD, we are not focused on testing. TDD is a tool to help us develop, not to help us test.

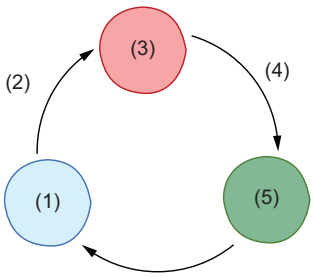
Let's revisit figure 1.4 in chapter 1. As I mentioned earlier, TDD is part of "testing to guide development." In other words, you should use TDD when you want your tests to guide you through the development process. When you are finished with your TDD sessions and the code looks good, it is time to begin the effective and systematic testing part of the process I describe: change your focus to testing, and apply specification-based testing, structural testing, and property-based testing.

Can you reuse the tests you created during TDD in the effective and systematic part of the process? Sure. Doing so becomes natural.

Combining TDD and effective testing makes even more sense when both are done in a timely manner. You do not want to TDD something and then wait a week before properly testing it. Can you mix short TDD cycles with short systematic and effective testing cycles? Yes! Once you master all the techniques, you will begin to combine them. The practices I discuss in this book are not meant to be followed linearly—they are tools that are always at your disposal.

Exercises

8.1 This figure illustrates the test-driven development cycle. Fill in the numbered gaps in the figure.



8.2 Which of the following is the least important reason to do TDD?

A TDD practitioners use the feedback from the test code as a design hint.

B The practice of TDD enables developers to have steady, incremental progress throughout the development of a feature.

C As a consequence of the practice of TDD, software systems are tested completely.

D Using mock objects helps developers understand the relationships between objects.

8.3 TDD has become a popular practice among developers. According to them, TDD has several benefits. Which of the following statements is *not* considered a benefit of TDD? (This is from the perspective of developers, which may not always match the results of empirical research.)

A Baby steps. Developers can take smaller steps whenever they feel it is necessary.

B Better team integration. Writing tests is a social activity and makes the team more aware of their code quality.

C Refactoring. The cycle prompts developers to improve their code constantly.

D Design for testability. Developers are forced to write testable code from the beginning.

8.4 It is time to practice TDD. A very common practice problem is calculating the final score of a bowling game.

In bowling, a game consists of 10 rounds. In each round, each player has a frame. In a frame, the player can make two attempts to knock over 10 pins with the bowling ball. The score for each frame is the number of pins knocked down, with a bonus for a strike or a spare.

A *strike* means the player knocks over all pins with one roll. In addition to 10 points for knocking down all 10 pins, the player receives a bonus: the total number of pins knocked over in the next frame. Here is an example: `[X] [1 2]` (each set of `[ ]` is one frame, and `X` indicates a strike). The player has accumulated a total of 16 points in these two frames. The first frame scores 10 + 3 points (10 for the strike, and 3 for the sum of the next two rolls, 1 + 2), and the second frame scores 3 (the sum of the rolls).

A *spare* means the player knocks down all pins in one frame, with two rolls. As a bonus, the points for the next roll are added to the score of the frame. For example, take `[4 /] [3 2]` (/ represents a spare). The player scores 13 points for the first frame (10 pins + 3 from the next roll), plus 5 for the second frame, for a total of 18 points.

If a strike or a spare is achieved in the tenth (final) frame, the player makes an additional one (for a spare) or two (for a strike) rolls. However, the total rolls for this frame cannot exceed three (that is, rolling a strike with one of the extra rolls does not grant more rolls).

Write a program that receives the results of the 10 frames and returns the game's final score. Use the TDD cycle: write a test, make it pass, and repeat.

Summary

- Writing a test that fails, making it pass, and then refactoring is what test-driven development is all about.
- The red-green-refactor cycle brings different advantages to the coding process, such as more control over the pace of development, and quick feedback.
- All the schools of TDD make sense, and all should be used depending on the current context.
- Empirical research does not find clear benefits from TDD. The current consensus is that working on small parts of a feature and making steady progress makes developers more productive. Therefore, while TDD is a matter of taste, using short implementation cycles and testing is the way to go.
- Deciding whether to use TDD 100% of the time is also a personal choice. You should determine when TDD makes you more productive.
- Baby steps are key to TDD. Do not be afraid to go slowly when you are in doubt about what to do next. And do not be afraid to go faster when you feel confident!