Learning Test-Driven Development by Counting Lines

Bas Vodde, *Nokia Networks*
Lasse Koskela, *Reaktor Innovations*

This training approach has students write tests that count lines of code. The problems they encounter provide valuable insights into TDD and its benefits.

During the last two years, Nokia Networks has started changing much of its product development from a traditional waterfall approach to Scrum, an agile software development process. Besides Scrum, there has been a lot of focus on engineering practices such as test-driven development.[1]

We've been involved in creating TDD training at Nokia Networks to support the transition to agile development. The first course was essentially a series of lectures. It wasn't considered particularly successful and had much less impact than expected.

So, we completely revised the training to involve mostly hands-on exercises. We also followed up with a weeklong TDD coaching session with the project staff, working on their actual code base. In effect, the training had transformed into two-day classroom sessions involving six exercises, accompanied by immediate follow-up to help the course participants apply the practice.

More than a dozen project teams have now received this training. During the training sessions, our first hands-on exercise—counting lines of code—proved exceptionally good for teaching TDD. Here we describe our experiences running this exercise and the most important things the participants and we learned from it.

The line-counting exercise

We derived this exercise from the Pragmatic Programmers' code kata 13.[2] Code katas are short exercises, some of which involve programming and can be coded in different ways.[3] They can be open ended, involving thinking about the issues behind programming.

Kata 13 asks the programmer to develop a piece of software that counts noncommented lines of Java code. In essence, the programmer should count lines that

- aren't empty and
- don't contain just comments.

Figure 1 shows an example of the desired line-counting logic, copied from a kata description.[2] As you can see from the left column in the figure, the file is considered to contain five lines of noncommented source code. The count ignores single-line, trailing, and multiline comments, along with any lines with only white space.

0740-7459/07/$25.00 © 2007 IEEE

```
-  /*****
-  * This is a test program with 5 lines of code
-  //*****//***/// Slightly pathological comment ending...
-
1 public class Hello {
2   public static void main(String [] args) { // comment
-       // Say hello
3     System./*wait*/out./*for*/println/*it*/("Hello/*");
4   }
5 }
```

Figure 1. An example of line-counting logic, copied from a kata description.

```
public class TestLineCounter extends TestCase {
  public void testFileWithOneLineOfCode() throws Exception {
    assertEquals(1, new LineCounter().count("OneLine.java"));
  }
}
```

Figure 2. A typical first test for a line counter.

The first nice thing about this exercise is its simplicity. Because lines of code is such a common concept in software development, none of our participants have had any difficulty understanding the problem. The other nice thing is that the problem isn't as simple as it seems. We'll get back to this aspect in a bit. But first, let's talk about how most participants initially approach their solution.

Revelations of the first test

The participants first need to decide where to start. Most participants learn fairly quickly that the first test should typically be very simple. For our line-counting exercise, however, what would be such a simple test?

We've observed many developers sketching their first test as something like figure 2. This test essentially aims to verify that the line counter returns the result of 1 for a source file with just one line. As such, a test for just one line is a nice low-hanging fruit—just the kind of test that we recommend starting with when you're new to TDD or when you hit the proverbial coder's block.

The test in figure 2 is also, however, an example of a common phenomenon we've observed among software developers. We're basically imposing an unnecessary, external dependency to the file system in the test. (Michael Feathers has written a good definition of a unit test in his blog.[4] One of his criteria is that the test must not touch the file system.)

Tests with such external dependencies are often an order of magnitude or two slower than ones without them. They're also somewhat less reliable than tests that, for example, work with in-memory data.

Because of the effort needed to set up source files somewhere in the file system and the additional explanation of why unit tests should avoid accessing the file system, most participants quickly end up using abstract streams or strings as input for the line counter class. Figure 3 illustrates two such designs.

After removing the external dependency to the file system, the participants move on to the next tests.

Making fast progress

After the first test, the participants start making fast progress. It's easy to add a new test and to find the simplest solution that could possibly work. In no time, we're looking at tests for a file with just one line of code, a file with two lines, lines of code with white space in between, and so forth.

At this point, a lot of duplication often accumulates in the participants' test code. This is because *refactoring*, the TDD cycle's third step, seems to be harder to remember than its predecessors "write a test" and "make it pass."

Refactoring the tests isn't difficult, though. What has proved to be a much more interesting problem to solve is the ugliness that arrives

```
String src = "this is one line of code";
InputStream stream = new ByteArrayInputStream(src.getBytes());
assertEquals(1, new LineCounter().count(stream));
```
(a)

```
String src = "this is one line of code";
assertEquals(1, new LineCounter().count(src));
```
(b)

Figure 3. Examples of the first test using a (a) stream-based API and (b) string-based API.

```
public class LineCounter {

  public void count(String code) {
    int counter = 0;
    String[] lines = code.split("\n");
    for (int i=0; i < lines.length; i++) {
      String line = lines[i].trim();
      if (line.equals("")) continue;
      if (line.startsWith("//")) continue;
      if (line.startsWith("/*") && line.endsWith("*/")) continue;
      ...
      counter++;
    }
    return counter;
  }
}
```

Figure 4. A sequence of guard clauses for handling simple constructs.

when participants extend their test sets so that the line count ignores multiline comments.

Ugliness arrives

For a line-counting test to ignore a source code snippet, it usually only has to process one line of code. However, the multiline C-style comment in our code example requires a slightly different approach. By the time participants write the first test for this comment, they've typically accumulated tests and the accompanying production code for a sequence of "guard clauses" inside some kind of a looping construct. Figure 4 attempts to describe a typical implementation pattern.

As participants get closer to writing a test for a multiline comment, they add more and more `if` statements one after another, and the code steadily becomes uglier. In the beginning, adding an `if` is trivial and possibly the simplest solution to the problem at hand, so participants do just that.

When the need to remember state arises in the form of the multiline-comment test, for most participants the obvious thing to do is introduce a flag variable and more `if` statements. Figure 5 depicts a typical implementation at this stage.

As we can see from figure 5, the implementation is getting increasingly complex, as nested branching constructs begin to appear and as keeping track of the business logic gets increasingly difficult.

At this stage, participants start to express dissatisfaction with their implementations. This is also where we as instructors often recommend stepping back and looking at the design—the big picture, if you will.

Stepping back to think

For some reason, aspiring developers learning TDD often seem to think that by following the cycle and always implementing the simplest-possible design, the perfect design will emerge—without them having to think about it. It's not quite that simple, however. TDD is a great way to develop software and can change the way you

```
public void count(String code) {
  int counter = 0;
  boolean insideMultilineComment = false;
  String[] lines = code.split("\n");
  for (int i=0; i < lines.length; i++) {
    String line = lines[i].trim();
    if (insideMultilineComment) {
      ... // bunch of if-statements omitted for brevity
    } else {
      ... // more if-statements omitted for brevity
    }
    counter++;
  }
  return counter;
}
```

Figure 5. Handling multiline comments using flags and even more `if` statements.

think about software and software development, but the developer's skill and confidence still play a big role in ensuring the outcome's quality.

We let participants struggle with the ugly code. The pain of adding new functionality slowly increases until it's next to impossible to figure out where and how to change the code to fix a newly added failing test while keeping the existing tests green (that is, they continue to pass). We point out the code's ugliness, and so far we haven't met a developer who disagrees with our assessment.

Some teams attempt simple refactorings, which makes their implementation read better but mostly just hides complexity behind descriptive "handles" such as variable and method names. Interestingly, of the dozens of participants we've observed facing this situation, not one ever stepped back to think about the design that has emerged.

At that point, some teams discuss at length how to refactor the implementation to exhibit less complexity. Most teams, however, try to write more tests, ignoring their solution's inherent ugliness. So, we have them reflect on their emergent design. We typically also point out how much worse their design would have become if they had proceeded in the same direction, because they would likely have added tests for source code that has different comment character sequences inside string literals.

The design that has emerged naturally through the tests they've written clearly isn't a sustainable path to follow. So what happened? What did they do wrong? Wasn't emergent design supposed to lead to better design?

The participants chose one design path at the exercise's beginning and never went back and asked themselves whether their design was still viable for their current needs. At a certain point, they need to look at their code and say, "Man, this looks ugly. We need a new approach."

A new perspective

As course instructors, we've done all the exercises ourselves. Like everyone else, we've ended up in the same situation with our line counter implementation getting increasingly complex. Doing this exercise and and reaching the limit of our patience with the ugly code we'd ended up with, we sat back and talked about our design on a higher level.

As we discussed what we had been doing, we gained a new perspective on the problem. We had started on the path of ignoring portions of the source code on the basis of a variety of patterns—with the accompanying drove of branching constructs making our code too ugly to bear. Our new perspective came from realizing that we had been essentially excluding blocks of code from getting processed.

At this point, we learned to formulate our design in new terms: removing stuff. We realized that we could simplify our design significantly by removing certain patterns such as single-line comments, empty lines, and multiline comments from the source code one at a time. After removing these structures, we would simply count the lines left in the source code. Figure 6 outlines this new, simpler design.

As you can see from figure 6, the new design turned out much easier to follow. (Well, at least

Figure 6. Removing patterns from the source code one at a time simplified the design significantly.

```
public class LineCounter {
  public int count(String code) {
    code = code.replaceAll("\r\n", "\n");
    code = code.replaceAll("( |\t)", "");
    code = code.replaceAll("(?s)/\\*.*?\n.*?\\*/", "\n");
    code = code.replaceAll("(?s)/\\*.*?\\*/", "");
    code = code.replaceAll("//[^\r\n]*", "");
    code = code.replaceAll("(?s)\n+", "\n");
    code = code.trim();
    return code.equals("") ? 0 : code.split("\n").length;
  }
}
```

after we gave meaningful names to all those cryptic replacement operations! We revised the design to save space.) We're not professional line-of-code-counter developers, and some other design might be a notch or two better still, but this design is already quite simple and serves its present purpose.

The big question

At this point, we were left with just one big question: could we have thought of this design up front? That is, could we have thought of all the different comment patterns and gotchas that our tests brought to light over time and devised a design that would have accommodated a clean implementation?

In our opinion, which we've shared with the participants, it's unlikely that we could have thought of this design without actually experiencing the pain of nested `if` statements and managing state. They all agreed.

We could imagine the final design only when we proceeded on the basis of our best understanding of the problem at the time, created one test at a time, learned about the problem, and proofed our design in practice. We couldn't have done this without writing code. We couldn't envision the problems we eventually encountered with our initial design until they raised their ugly heads. The design had to emerge through the experience of attempting to implement it.

Interestingly, changing our design halfway through the process wasn't at all difficult, even though the new approach differed significantly from the original. The key to this flexibility was that the code was well factored and didn't suffer from duplication. All the tests were still there to verify that the design change didn't break any existing functionality.

You could argue that we might have attempted to bite the bullet with our old design for quite a while longer if we hadn't been practicing TDD. Following TDD and refactoring our code mercilessly, we had to face the facts and do something about the poor design. We've concluded that the line-counting exercise is extremely helpful in providing insights into TDD and its benefits, including these:

- Removing external dependencies helps improve testability.
- Reflective thinking promotes emergent design.
- A well-factored design and good test coverage also help new designs emerge.

For us, as TDD trainers, the line-counting

About the Authors

Bas Vodde led the change to agile development at Nokia Networks (now Nokia Siemens Networks). He's had a variety of roles in both traditional large projects and agile projects at Nokia and elsewhere. His major interests are in software development, quality management, lean thinking, and agile methods—in particular, Scrum. He also now and then works for his own company, called odd-e. Contact him at bas.vodde@nsn.com.

Lasse Koskela is a methodology specialist at Reaktor Innovations. He has also held roles varying from development to project management and from training to consulting, dipping his toes in sales every once in a while. He's the author of a book on test-driven development, to be published in 2007 by Manning Publications. Contact him at Reaktor Innovations, Mannerheimintie 2, FIN-00100 Helsinki, Finland; lasse.koskela@ri.fi.

exercise's role has changed over time. In the beginning, we considered it just a nice, simple first exercise. But because it has revealed many key insights into TDD, it has become one of the most important exercises in the training.

We thank all the course participants for their feedback. The power of the simple exercise we've described here is just one of the things we've learned while training them.

References

1. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley Professional, 2002.
2. D. Thomas, "Kata Thirteen: Counting Code Lines," 31 July 2003, http://blogs.pragprog.com/cgi-bin/pragdave.cgi/Practices/Kata/KataThirteen.rdoc.
3. D. Thomas, "Code Kata," 13 Oct. 2004, http://codekata.pragprog.com/codekata.
4. M. Feathers, "A Set of Unit Testing Rules," 9 Sept. 2005, www.artima.com/weblogs/viewpost.jsp?thread=126923.