The Productive Programmer

Neal Ford

foreword by David Bock

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

The Productive Programmer
by Neal Ford

Copyright © 2008 Neal Ford. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Loranah Dimant
Copyeditor: Emily Quill
Proofreader: Loranah Dimant

Indexer: Fred Brown
Cover Designer: Mark Paglietti
Interior Designer: David Futato
Illustrator: Robert Romano
Photographer: Candy Ford

Printing History:
July 2008: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51978-0

[C]

1213991395

CONTENTS

FOREWORD vii

PREFACE ix

1 INTRODUCTION 1

Why a Book on Programmer Productivity? 2

What This Book Is About 3

Where to Go Now? 5

Part One MECHANICS

2 ACCELERATION 9

Launching Pad 10

Accelerators 18

Macros 33

Summary 35

3 FOCUS 37

Kill Distractions 38

Search Trumps Navigation 40

Find Hard Targets 42

Use Rooted Views 44

Use Sticky Attributes 46

Use Project-Based Shortcuts 47

Multiply Your Monitors 48

Segregate Your Workspace with Virtual Desktops 48

Summary 50

4 AUTOMATION 51

Don't Reinvent Wheels 53

Cache Stuff Locally 53

Automate Your Interaction with Web Sites 54

Interact with RSS Feeds 54

Subvert Ant for Non-Build Tasks 56

Subvert Rake for Common Tasks 57

Subvert Selenium to Walk Web Pages 58

Use Bash to Harvest Exception Counts 60

Replace Batch Files with Windows Power Shell 61

Use Mac OS X Automator to Delete Old Downloads 62

Tame Command-Line Subversion 62

Build a SQL Splitter in Ruby 64

Justifying Automation 65

Don't Shave Yaks 67
Summary 68

5 CANONICALITY 69
DRY Version Control 70
Use a Canonical Build Machine 72
Indirection 73
Use Virtualization 80
DRY Impedance Mismatches 80
DRY Documentation 88
Summary 93

Part Two PRACTICE

6 TEST-DRIVEN DESIGN 97
Evolving Tests 99
Code Coverage 105

7 STATIC ANALYSIS 109
Byte Code Analysis 110
Source Analysis 112
Generate Metrics with Panopticode 113
Analysis for Dynamic Languages 116

8 GOOD CITIZENSHIP 119
Breaking Encapsulation 120
Constructors 121
Static Methods 121
Criminal Behavior 126

9 YAGNI 129

10 ANCIENT PHILOSOPHERS 135
Aristotle's Essential and Accidental Properties 136
Occam's Razor 137
The Law of Demeter 140
Software Lore 141

11 QUESTION AUTHORITY 143
Angry Monkeys 144
Fluent Interfaces 145
Anti-Objects 147

12 META-PROGRAMMING 149
Java and Reflection 150
Testing Java with Groovy 151
Writing Fluent Interfaces 152
Whither Meta-Programming? 154

13 COMPOSED METHOD AND SLAP 155
Composed Method in Action 156

SLAP 160

14 POLYGLOT PROGRAMMING 165
How Did We Get Here? And Where Exactly Is Here? 166
Where Are We Going? And How Do We Get There? 169
Ola's Pyramid 173

15 FIND THE PERFECT TOOLS 175
The Quest for the Perfect Editor 176
The Candidates 179
Choosing the Right Tool for the Job 180
Un-Choosing the Wrong Tools 186

16 CONCLUSION: CARRYING ON THE CONVERSATION 189

APPENDIX: BUILDING BLOCKS 191

INDEX 199

CHAPTER SIX

Test-Driven Design

UNIT TESTING IS WELL ESTABLISHED AS A BENEFICIAL CODE-HYGIENE PRACTICE. Tested code provides greater confidence that the intent matches the result. Test-driven development (TDD) goes even further, insisting that you write tests before you write the code. When comparing software "engineering" to other engineering disciplines (which always requires a good handful of tortured metaphors), major differences pop up. We don't have centuries of mathematics to rely upon in software. The science of software development hasn't been around long enough (and we may never get to that level of sophistication). We also can't take advantage of the economy of scale in traditional engineering. For example, the Golden Gate Bridge contains more than 1,000,000 rivets. You can bet that the engineers who designed that bridge knew the stress characteristics of those rivets and used that number multiplied by 1,000,000 to tell them important things about the stress on the bridge. A piece of software may also have 1,000,000 pieces, but they are all different. We can't take advantage of the scale and multiplicity that "regular" engineers can. But developers do have an advantage: we can manufacture our components very easily and write code that verifies our components do what we intended them to do. Because it is vanishingly cheap to write software to test software, we can apply our own version of verification through levels of testing: unit testing, functional testing, integration testing, and user acceptance.

> NOTE
> **Testing *is* the engineering rigor of software development.**

Rigorously applied TDD has other design benefits as well, so many that I usually refer to TDD as test-driven *design*. TDD forces you to think about code in a different way. Instead of just writing a pile of code and then writing the tests for it, TDD forces you to think through the testing process before you write the code. TDD creates *consumption awareness*: when you create a unit test, you are creating the first consumer of the code under development. This forces you to think about how the rest of the world will use this class. Every developer has the experience of writing a class in one big bunch, making assumptions along the way. Then, when it comes time to actually use the class, you realize that some of your assumptions were wrong, and you have to refactor the original code. TDD requires that you create the first consumer before you write the code, making you think about how other code will eventually use the code under development.

TDD also forces you to mock out dependent objects. For example, if you develop a `Customer` class with an `addOrder` method, you must collaborate with an `Order` object. If you create the dependent object within the `addOrder` method, it requires that the `Order` object exist before you can test the `Customer` class. Mock objects allow you to create a "fake" version of the dependent class for testing purposes. You must think about the interaction between the two objects at exactly the right time: as you are developing the first of the two classes.

TDD encourages you to pass dependent objects via fields or parameters, leaving the construction of the dependent objects elsewhere (because you can't mock out a dependency if the method fires the constructor itself). This tends to move object construction to well-defined boundary layers, making it easier to track allocation and referencing (so that you don't inadvertently hold accidental references to objects, preventing them from garbage collection because they never go out of scope). TDD also virtually forces you to have very small, cohesive methods because you must write tests that test only one thing. Your methods tend to do only one thing as well, making them adhere more closely to the SLAP principle (covered in Chapter 13).

Evolving Tests

Let's look at an example of the design benefits that TDD affords. To demonstrate them, we need a problem that isn't so trivial as to be a throwaway, but not so complex that we get caught up in the details. A perfect candidate (pun intended) is a *perfect number* finder. A perfect number is a number whose factors (minus the number itself) add up to the number. For example, 6 is a perfect number because the factors of 6 (1, 2, 3, and 6) minus the 6 add up to 6. Let's write some code in Java to find perfect numbers.

TDDing Unit Tests

The following code was written without TDD, just by applying simple logic and some minor mathematical optimizations:

```
public class PerfectNumberFinder {

    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < Math.sqrt(number) + 1; i++)❶
            if (number % i == 0) {
                factors.add(i);
                if ( number / i != i )❷
                    factors.add(number / i);
            }

        // sum the factors
        int sum = 0;
        for (Integer i : factors)
            sum += i;

        // decide if its perfect
        return sum - number == number;
    }
}
```

❶ If you can harvest the numbers in pairs, you only have to go up to the square root of the number. For example, for the number 28, when you find the factor 2, you can also harvest the number 14, the symmetrical factor.

❷ The code `number / i != i` is present to make sure that you don't add the same number to the list twice. Because you're harvesting symmetrical pairs, what happens when the factor is the same? For example, in the case of 16, when you get the factor 4, you'll need to add it to the list only once.

This code has a single static method that returns true or false depending on the perfection of the passed number. The first step gets the factors. I know that 1 and the number itself are factors, so I add them right away. Then, I use a `for` loop to go up to the square root of the number. This is a slight optimization; if you harvest the factors in pairs, you only have to search up to the square root.

As it stands, this is just a blob of code. How would it look different if TDD were used? The first test should be almost insanely simple. Here, I just want to get the factors for 1:

```
@Test public void factors_for_1() {
    int[] expected = new int[] {1};
    Classifier c = new Classifier(1);
    assertThat(c.getFactors(), is(expected));
}
```

I'm using JUnit 4.4 with the Hamcrest* matchers (Hamcrest matchers provide more English-friendly syntax for matchers, such as `assertThat(expected, is(c.getFactors()))`). How can this be a useful test? It seems too simple. Really simple tests like this aren't really about testing, they are about getting the infrastructure set up correctly. I must have the test libraries on the classpath, I need a class named `Classifier`, and I must resolve all the package dependencies. That's a lot of work! Writing a stupidly simple test allows me to get all the structure established before I have to start thinking about testing the actual hard problems.

Once I get this test to pass, I enhance it a little to make it look more like the expected real tests, changing it to a resizable list via a `List<Integer>`:

```
@Test public void factors_for_1() {
    List<Integer> expected = new ArrayList<Integer>(1);
    expected.add(1);
    Classifier c = new Classifier(1);
    assertThat(c.getFactors(), is(expected));
}
```

Once I get this test to pass, should I keep it around? Yes! I call these really simple tests *canary tests*. Just as miners took canaries into coal mines to warn of encroaching gas, this test performs a constant reality check for your tests. If it ever fails, you have serious problems with the infrastructure of your code: a JAR file has gotten misplaced, the code itself has moved, etc. These very simple tests can tell you if something fundamental has broken.

* Download at *http://code.google.com/p/hamcrest/*.

The next test I want to undertake checks for real factors of a number:

```
@Test public void factors_for_6() {
    List<Integer> expected = new ArrayList<Integer>(
        Arrays.asList(1, 2, 3, 6));
    Classifier c = new Classifier(6);
    assertThat(c.getFactors(), is(expected));
}
```

This is the test I want to write, but it represents lots of different functionality: to make this test pass, I must know if a number is a factor, how to calculate factors, and how to harvest the factors I've found. This happens frequently during the TDD process: one test reveals lots of desired functionality. The best way to attack this daunting pile of work is to step back and think about what it takes to make this test a reality. By drilling down, I create the following tests (and the corresponding code to make them pass):

```
@Test public void is_factor() {
    assertTrue(Classifier.isFactor(1, 10));
    assertTrue(Classifier.isFactor(5, 25));
    assertFalse(Classifier.isFactor(6, 25));
}

@Test public void add_factors() {
    Classifier c = new Classifier(20);
    c.addFactor(2);
    c.addFactor(4);
    c.addFactor(5);
    c.addFactor(10);
    List<Integer> expectation = new ArrayList<Integer>(
            Arrays.asList(1, 2, 4, 5, 10, 20));
    assertThat(c.getFactors(), is(expectation));
}
```

For the `Classifier` population code, `1` and the number itself (`20`) are added automatically, so I add the remaining factors. The first test is fine, but the second test fails after I implement the code in `Classifier` to populate the `ArrayList`:

```
java.lang.AssertionError: Expected: is <[1, 2, 4, 5, 10, 20] got: <[1, 20, 2, 10, 4, 5]>
```

The unexpected result stems from harvesting my numbers in pairs. This raises a fundamental question: should I add code to my `Classifier` to prevent duplication, or am I using the wrong abstraction? Factors have no inherent order, they are really a set. That indicates that I should change my `Classifier` to use a `HashSet` instead of an `ArrayList`. TDD excels at helping find mistaken assumptions early, where the pain of refactoring is small because there isn't much code yet. One interesting side note to the testing example: the `addFactor()` method is actually `private` in `Classifier`. I show how to test these kinds of private methods in "Java and Reflection" in Chapter 12.

Following this process through to its logical conclusion yields the following implementation of `Classifier`:

```
public class Classifier {
    private int _number;
    private Set<Integer> _factors;
```

```
    public Classifier(int number) {
        if (number < 0) throw new InvalidNumberException();
        setNumber(number);
    }

    public Classifier() {}

    public Set<Integer> getFactors() {
        return _factors;
    }

    public boolean isPerfect() {
        return sumOfFactorsFor(_number) - _number == _number;
    }

    public void calculateFactors() {
        for (int i = 2; i < Math.sqrt(_number) + 1; i++)
            addFactor(i);
    }

    private void addFactor(int i) {
        if (isFactor(i)) {
            _factors.add(i);
            _factors.add(_number / i);
        }
    }

    private int sumOfFactorsFor(int number) {
        calculateFactors();
        int sum = 0;
        for (int i : _factors)
            sum += i;
        return sum;
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }

    public int getNumber() {
        return _number;
    }

    public void setNumber(int value) {
        _number = value;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
    }
}
```

Measurements

If you compare the TDD version of the code to the non-TDD version, you'll see that the TDD one has much more code, but in lots of really small methods. Lots of small methods are good. If you read the method names, you will get a strong sense of the atomic operations required to compute perfect numbers. In fact, if you look at the comments in the original code, the same functionality (plus some) will show up as methods in the TDD version.

In six months, when it comes time to alter this code, you can make changes with confidence. If something breaks, you will know within a few lines of code what's wrong. The method names in TDD code describe an atomic operation, so when a test fails, you understand what you've broken more quickly. When dealing with code that has much longer methods, narrowing down to the errors takes much longer because you have to understand the context of the entire method before you can make changes to it. Understanding a three-line method shouldn't take any time at all. If you find yourself writing embedded comments within methods, your method should be more refined. Long methods with lots of comments reek of a noncomposed solution. Refactor the comments to methods to get rid of them.

> **NOTE**
> **Refactor comments to methods.**

One of the (few) useful code metrics is McCabe's Cyclomatic Complexity (see the next sidebar "Cyclomatic Complexity"). The average cyclomatic complexity for the `PerfectNumberFinder` class (the first, non-TDD version) is 5 (which is the cyclomatic complexity for the lone method, thus the average for the class). The TDD version scores a class average cyclomatic complexity of 1.5, which indicates that its methods (and therefore the class) are much simpler.

CYCLOMATIC COMPLEXITY

Thomas McCabe created the code metric called Cyclomatic Complexity to measure the complexity of code. The formula is quite simple: the number of edges – the number of nodes + 2, where the edges represent the execution path and nodes represent lines of code. For example, consider the following:

```
public void doit() {
    if (c1) {
      f1();
    } else {
      f2();
    }
    if (c2) {
      f3();
    } else {
      f4();
    }
}
```

If you graph this method like a flow chart (see Figure 6-1), you'll discover 8 edges and 7 nodes, meaning that this code has a cyclomatic complexity of 3.

Many tools exist to determine cyclomatic complexity (see Chapter 7 for some open source ones), including the analysis menus in the IntelliJ IDE.

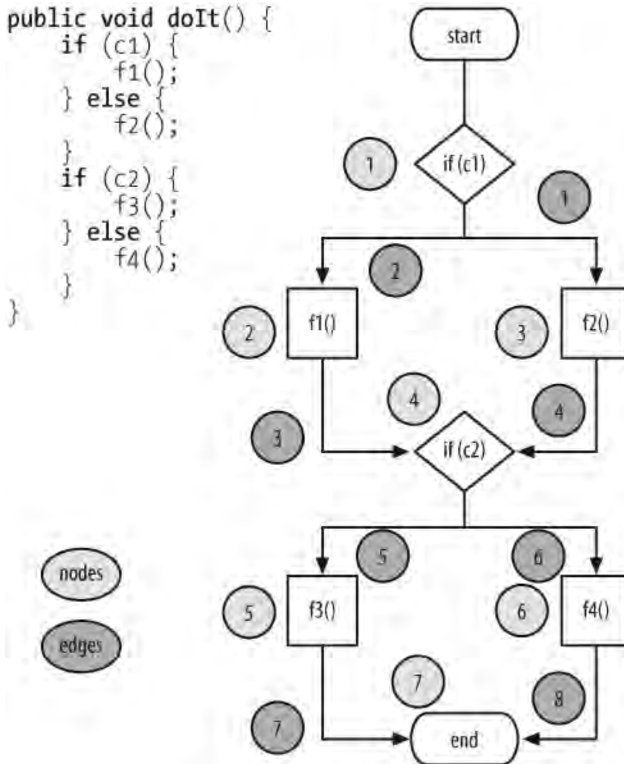

FIGURE 6-1. Determining cyclomatic complexity with a flow chart

Design Impact

Design impact is one last indicator of the design benefits of TDD. Let's say that your insatiable users pop up and decide that they want not only a perfect number finder, but one that calculates abundant numbers (numbers whose factors sum to more than the number) and deficient numbers (numbers whose factors sum to less than the number). In the first version of the perfect number finder, you'd have to go on a massive refactoring expedition, breaking it apart into something resembling the second version. And for the TDD version? Just add two methods:

```
public boolean isDeficient() {
    return sumOfFactorsFor(_number) - _number < _number;
}

public boolean isAbundant() {
    return sumOfFactorsFor(_number) - _number > _number;
}
```

All the building blocks already exist. TDD code tends to have more reusable elements because it forces you to write ultra-cohesive methods, and cohesive methods are the building blocks for truly reusable code.

TDD improves the design of your code, providing the following benefits:

- It forces consumption awareness in your code because you create the first consumer before you create any code.
- Testing (and keeping the tests) for extremely trivial initial cases provides warnings when you've accidentally broken critical infrastructure.
- Testing edge cases and boundary conditions is essential. Things that are difficult to test should either be refactored into simpler things, or, if you can't simplify them, they should be rigorously tested, no matter how hard it is. Complex things need testing the most!
- Always keep all your tests as part of your build process. The most insidious things in software are the side-effect faults that you accidentally introduce when making changes to a completely unrelated chunk of code. Running your unit tests as regression tests allows you to find those side effects immediately. Having this safety net of unit tests always saves you time and effort.
- Having a strong set of unit tests allows you to play "what if" refactoring games (that is, you can make a broad change and run your tests to determine the impact). When I first started working with developers who were accustomed to having strong unit tests, they would start making changes to the code, which made me nervous because wholesale changes can break lots of stuff. But they would do it at the drop of a hat, and I eventually came around when I realized that having tests gives you confidence to make changes that improve your code.

Code Coverage

One last really important testing topic is *code coverage*, which refers to the lines and branches executed in your code by tests. Open source and commercial code coverage tools exist for virtually every language.

For compiled languages (like Java and C#), code coverage works by first running an instrumentation processor on your compiled byte code. You then run your suite of unit tests against the instrumented code, which measures which lines are executed. The details are written to some intermediate form, from which a report is generated, showing the line and branch coverage of your tests. This is summarized in Figure 6-2.

The process is slightly different for dynamic languages, but the end result is the same: a report on how much of your code has been exercised by your tests. The report appears in your IDE, or in an XML or HTML view.

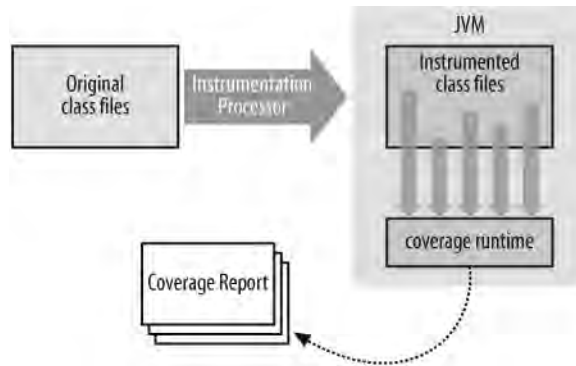

FIGURE 6-2. How code coverage works

This metric is critical because it tells you what has not been tested. Testing is the engineering rigor of software, and untested code is most likely where bugs live. If you are religious about TDD, all your code will be tested automatically, except for unusual fringe cases (for which you should add tests).

A lot of developers ask what the acceptable level of code coverage is. I used to be sanguine about this number, setting the acceptable threshold at about 80 percent. Then I noticed an interesting phenomenon: accepting only 80 percent coverage meant that the code that needed testing the most didn't get tests. Even conscientious developers write some complex code, run the code coverage report, and say "Whew! 82.3 percent. I wasn't looking forward to figuring out how to test that beast!"

I've come to the conclusion that anything less than 100 percent code coverage is a dangerous compromise. If you hold yourself to that highest standard, you will never have any code that's "too complex to test." You must write simpler code and when you encounter a truly complex scenario, you will be forced to come up with innovative ways to test it. Knowing that you don't have a "get out of testing free" card will make you more diligent about code hygiene.

But what if you already have a large code base that has no tests? It is wildly impractical to imagine that you can stop active development for months just to shore up your code coverage. First, set a date, somewhere in the near future (like next Thursday). Then, get the entire development team to agree that, after the inception date, your code coverage will always increase. That means that:

- All new code gets unit tests with 100 percent coverage (hopefully developed by TDD).
- Every time you fix a bug, you write a test.

It takes a great deal of effort to achieve 100 percent code coverage. You will write tests for all new code (which keeps it simpler), and you will write tests when you find bugs, meaning that the likelihood of bugs will diminish.

Like I just said, 100 percent code coverage on unit tests is a hard standard to achieve. However, I've been on projects that manage it, and it invariably improves the objective characteristics of the code (measured using static analysis and other measures; see Chapter 7).