WORKING EFFECTIVELY WITH UNIT TESTS

Jay Fields

Working Effectively with Unit Tests

Jay Fields

This book is for sale at http://leanpub.com/wewut

This version was published on 2014-07-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Jay Fields by spreading the word about this book on Twitter!

The suggested hashtag for this book is #wewut.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#wewut

For Dana, the love of my life.

Contents

Preface

Over a dozen years ago I read *Refactoring*[1] for the first time; it immediately became my bible. While *Refactoring* isn't about testing, it explicitly states: If you want to refactor, the essential precondition is having solid tests. At that time, if *Refactoring* deemed it necessary, I unquestionably complied. That was the beginning of my quest to create productive unit tests.

Throughout the 12+ years that followed reading *Refactoring* I made many mistakes, learned countless lessons, and developed a set of guidelines that I believe make unit testing a productive use of programmer time. This book provides a single place to examine those mistakes, share the lessons learned, and provide direction for those looking to test in a way that I've found to be the most effective.

Why Test?

The answer was easy for me: *Refactoring* told me to. Unfortunately, doing something strictly because someone or something told you to is possibly the worst approach you could take. The more time I invested in testing, the more I found myself returning to the question: Why am I writing this test?

There are many motivators for creating a test or several tests:

- validating the system
  - immediate feedback that things work as expected
  - prevent future regressions
- increase code-coverage
- enable refactoring
- document the behavior of the system
- your manager told you to
- Test Driven Development
  - improved design
  - breaking a problem up into smaller pieces
  - defining the "simplest thing that could possibly work"
- customer acceptance
- ping pong pair-programming

Some of the above motivators are healthy in the right context, others are indicators of larger problems. Before writing any test, I would recommend deciding which of the above are motivating

[1] http://martinfowler.com/books/refactoring.html

you to write a test. If you first understand why you're writing a test, you'll have a much better chance of writing a test that is maintainable and will make you more productive in the long run.

Once you start looking at tests while considering the motivator, you may find you have tests that aren't actually making you more productive. For example, you may have a test that increases code-coverage, but provides no other value. If your team requires 100% code-coverage, then the test provides value. However, if your team has abandoned the (in my opinion harmful) goal of 100% code-coverage, then you're in a position to perform my favorite refactoring: delete.

Who Should Read This Book

This book is aimed at a professional programmer, someone who writes software for a living. The examples and discussion include a lot of code to read and to understand.

Although this book is focused on testing, testable code can have a large impact on the design of a system. It is vital for senior designers and architects to understand the principles recommended and to use them in their projects. The principles within this book are best introduced to a team by a respected and experienced developer. Such a developer can best understand the principles and adapt them to their specific context. In addition, familiarity with this book's content will allow experienced developers to provide it as a reference for the less experienced members of their team.

Despite my opinion on who should introduce these concepts, I've attempted to write this book for both people experienced with and those brand-new to unit testing. Ideally, a respected programmer will look to implement the ideas within this book, and begin by passing this book on to those on the team that would likely share interest in this approach. If you're already writing tests I believe this book will provide a foundation that will help you productively test your applications for the entirety of your career. If you aren't already writing tests you'll likely want to pick up an intro to unit testing book as well. The concepts in this book should be understandable to developers of all levels; however, we will not cover concepts such as framework selection, framework configuration, or writing your first test.

Building on the Foundations Laid by Others

While this book does contain an Acknowledgments section, it wouldn't be practical to thank everyone that has contributed to creating the practices that this book details. I can say with full confidence that I wouldn't be in the position to write this book without at least the following groups:

- creators and maintainers of both NUnit and JUnit
- creators and maintainers of NMock, (James Mead's) Mocha, Mockito, JMock, and RSpec
- each team member from each of my projects at ThoughtWorks & DRW Trading
- every conference speaker or attendee who's provided feedback on my radical ideas
- every person who took the time to comment on http://blog.jayfields.com[2]

Thank you all, I deeply appreciate the feedback you've given throughout the years.

[2] http://blog.jayfields.com

Acknowledgments

After writing *Refactoring: Ruby Edition*, I swore I'd never write another book. Book writing is unquestionably a labor of love, and I wouldn't be able to do it without the support of my many friends in the industry.

- Martin Fowler: Thank you for allowing me to reference and reuse content from *Refactoring*. It's still my favorite technical book of all time.
- Obie Fernandez: Thank you for the nudge to use leanpub; it was crucial for making this project happen.
- Michael Feathers: Honestly, I just liked the way *Working Effectively with Unit Tests* sounded. I never considered that anyone would associate this book with a book as universally loved as *Working Effectively with Legacy Code*. Nonetheless, I'll do my best to deliver a book that is worthy of being on the same shelf as *Working Effectively with Legacy Code*. Thank you very much for your blessing.
- Original Reviewers: There's no question this book is significantly better due to the feedback I got from those who originally volunteered to provide feedback. Thank you David Chelimsky, Graham Nash, John Hume, Pat Farley, & Steve McLarnon.

Additionally, I've been happily surprised by the support I've gotten from people who purchased the early edition on leanpub and promptly provided feedback. Many thanks - Corey Haines, J. B. Rainsberger, Jake McCrary, Josh Graham, Kent Spillner.

I'm sure there are others who I've forgotten; I apologize and offer my thanks.

Unit Testing, a First Example

I'd like to begin this book with an example, and I believe Martin's description of why is as clear as it can be written:

> Traditionally technical books start with a general introduction that outlines things like history and broad principles. When someone does that at a conference, I get slightly sleepy. My mind starts wandering with a low-priority background process that polls the speaker until he or she gives an example. The examples wake me up because it is with examples that I can see what is going on. With principles it is too easy to make generalizations, too hard to figure out how to apply things. An example helps make things clear. –Martin Fowler, Refactoring: Ruby Edition

Note: If the following domain looks familiar to you, that's because I've borrowed it from *Refactoring*.

Without further ado, I present a test failure.

```
JUnit version 4.11
.E.E..
There were 2 failures:
1) statement(CustomerTest)
org.junit.ComparisonFailure: expected:<...or John
        Godfather 4[        ]9.0
Amount owed is 9...> but was:<...or John
        Godfather 4[ ]9.0
Amount owed is 9...>
2) htmlStatement(CustomerTest)
org.junit.ComparisonFailure: expected:<...</h1>
<p>Godfather 4[        ]9.0</p>
<p>Amount ow...> but was:<...</h1>
<p>Godfather 4[ ]9.0</p>
<p>Amount ow...>

FAILURES!!!
Tests run: 4,  Failures: 2
```

The above output is what JUnit will report (sans stacktrace noise) for the (soon to follow) `CustomerTest` class.

Unless you work alone and on greenfield projects exclusively, you'll often find yourself looking at a test for the first time when it's reporting a failure. If that's a common case you'll encounter at work then it feels like a great way to start the book as well.

Below you'll find the cause of the failure, the `CustomerTest` class.

```
public class CustomerTest {
  Customer john, steve, pat, david;
  String johnName = "John",
    steveName = "Steve",
    patName = "Pat",
    davidName = "David";
  Customer[] customers;

  @Before
  public void setup() {
    david = ObjectMother
      .customerWithNoRentals(
        davidName);
    john = ObjectMother
      .customerWithOneNewRelease(
        johnName);
    pat = ObjectMother
      .customerWithOneOfEachRentalType(
        patName);
    steve = ObjectMother
      .customerWithOneNewReleaseAndOneRegular(
        steveName);
    customers =
      new Customer[]
      { david, john, steve, pat};
  }

  @Test
  public void getName() {
    assertEquals(
      davidName, david.getName());
    assertEquals(
      johnName, john.getName());
    assertEquals(
      steveName, steve.getName());
    assertEquals(
      patName, pat.getName());
```

```
  }

  @Test
  public void statement() {
    for (int i=0; i<customers.length; i++) {
      assertEquals(
        expStatement(
          "Rental record for %s\n" +
          "%sAmount owed is %s\n"  +
          "You earned %s frequent " +
          "renter points",
          customers[i],
          rentalInfo(
            "\t", "",
            customers[i].getRentals())),
        customers[i].statement());
    }
  }

  @Test
  public void htmlStatement() {
    for (int i=0; i<customers.length; i++) {
      assertEquals(
        expStatement(
          "<h1>Rental record for " +
          "<em>%s</em></h1>\n%s" +
          "<p>Amount owed is <em>%s</em>" +
          "</p>\n<p>You earned <em>%s" +
          " frequent renter points</em></p>",
          customers[i],
          rentalInfo(
            "<p>", "</p>",
            customers[i].getRentals())),
        customers[i].htmlStatement());
    }
  }

  @Test
  (expected=IllegalArgumentException.class)
  public void invalidTitle() {
    ObjectMother
      .customerWithNoRentals("Bob")
```

```
      .addRental(
        new Rental(
          new Movie("Crazy, Stupid, Love.",
                    Movie.Type.UNKNOWN),
          4));
  }

  public static String rentalInfo(
    String startsWith,
    String endsWith,
    List<Rental> rentals) {
    String result = "";
    for (Rental rental : rentals)
      result += String.format(
        "%s%s\t%s%s\n",
        startsWith,
        rental.getMovie().getTitle(),
        rental.getCharge(),
        endsWith);
    return result;
  }

  public static String expStatement(
    String formatStr,
    Customer customer,
    String rentalInfo) {
    return String.format(
      formatStr,
      customer.getName(),
      rentalInfo,
      customer.getTotalCharge(),
      customer.getTotalPoints());
  }
}
```

The `CustomerTest` class completely covers our `Customer` domain object and has very little duplication; many would consider this a well written set of tests.

As you can see, we're using an ObjectMother to create our domain objects. The following code represents the full definition of our `ObjectMother`.

```
public class ObjectMother {
  public static Customer
  customerWithOneOfEachRentalType(
    String name) {
    Customer result =
      customerWithOneNewReleaseAndOneRegular(
        name);
    result.addRental(
      new Rental(
        new Movie("Lion King", CHILDREN), 3));
    return result;
  }

  public static Customer
  customerWithOneNewReleaseAndOneRegular(
    String n) {
    Customer result =
      customerWithOneNewRelease(n);
    result.addRental(
      new Rental(
        new Movie("Scarface", REGULAR), 3));
    return result;
  }

  public static Customer
  customerWithOneNewRelease(
    String name) {
    Customer result =
      customerWithNoRentals(name);
    result.addRental(
      new Rental(
        new Movie(
          "Godfather 4", NEW_RELEASE), 3));
    return result;
  }

  public static Customer
  customerWithNoRentals(String name) {
    return new Customer(name);
  }
}
```

Thoughts on our Tests

Our `CustomerTest` class is written in a way that follows many common patterns. It doesn't take much searching on the Web to find articles giving examples of "improving" your code by using a Setup (now `@Before` in JUnit). ObjectMother lives under many names, and each name comes with several articles explaining how it's either successful or the programmer didn't understand how to correctly use the pattern. Finally, our tests follow the common advice that above all, code must be DRY[3].

> DRY is an acronym for Don't Repeat Yourself, and is defined as: *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Each of those pieces of advice are contextually valuable. I can easily think of situations where applying each of those patterns would be the right choice. However, in the context of "I would like to quickly understand this test I've never seen before", those patterns come up short. While working on code written by a teammate or supporting an inherited system, I find myself in the latter context far more often than not.

I suspect most people will have skimmed the above tests - that's what I would have done. Other people may have taken the time to try to understand the test and how it relates to the failure output. If you're in the second group, I suspect your thought process might have looked something like this.

1. find the `statement` test
2. find the definition of the `customers` array that we're iterating
3. find the assignment to `customers`
4. digest the assignment of each `Customer` and their associated name
5. look to `ObjectMother` to determine how the `Customer` instances are created
6. digest each of the different `Customer` instance creation methods within the `ObjectMother`
    - you now understand the first line of the test
7. digest that the expected value is being created by calling a method with a `String`, a `Customer`, and the result of calling `rentalInfo` with 2 `String` instances and a customer's `rentals`.
8. find the `rentalInfo` method and determine what value it's returning to `expStatement`
9. digest that `rentalInfo` is creating a string by iterating and formatting `Rental` data
10. now that you've mentally resolved the args to `expStatement`, you find that method and digest it.
    - at this point it's taken 10 steps to simply understand the expected value in your test
11. recognize that the actual value is a call to the domain object, who's source I haven't supplied (yet).

[3] http://nat.truemesh.com/archives/000714.html

That's quite a bit you needed to digest, and all of it test code. Not one character of what you've digested will actually run in production.

Were you actually trying to fix this test, the next logical question would be: Which is incorrect, the expected value or the actual value? Unfortunately, before you could even begin to tackle that question you'd need to find out what the expected and actual values actually are. We can see the text differs around the word "Godfather", but that only narrows our list down to the customers `john`, `steve`, and `pat`. It's practically impossible to fix this test without writing some code to help you debug the issue.

The Domain Code

It's not necessary to digest the domain code to complete this chapter.

> note: below is exactly what the domain code from *Refactoring* would look like if it were written in Java 7.

I would recommend skimming or completely skipping to the end of this section, and coming back to use this as a reference only if you're truly curious about the details of the domain.

```
public class Customer {

  private String name;
  private List<Rental> rentals =
    new ArrayList<Rental>();

  public Customer(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public List<Rental> getRentals() {
    return rentals;
  }

  public void addRental(Rental rental) {
    rentals.add(rental);
  }

  public String statement() {
    String result =
      "Rental record for " + getName() + "\n";
    for (Rental rental : rentals)
      result +=
        "\t" + rental.getLineItem() + "\n";
    result +=
      "Amount owed is " + getTotalCharge() +
      "\n" + "You earned " +
      getTotalPoints() +
```

```
      " frequent renter points";
    return result;
  }

  public String htmlStatement() {
    String result =
      "<h1>Rental record for <em>" +
      getName() + "</em></h1>\n";
    for (Rental rental : rentals)
      result += "<p>" + rental.getLineItem() +
        "</p>\n";
    result +=
      "<p>Amount owed is <em>" +
      getTotalCharge() + "</em></p>\n" +
      "<p>You earned <em>" +
      getTotalPoints() +
      " frequent renter points</em></p>";
    return result;

  }

  public double getTotalCharge() {
    double total = 0;
    for (Rental rental : rentals)
      total += rental.getCharge();
    return total;
  }

  public int getTotalPoints() {
    int total = 0;
    for (Rental rental : rentals)
      total += rental.getPoints();
    return total;
  }

}
```

```
public class Rental {

  Movie movie;
  private int daysRented;

  public Rental(Movie movie, int daysRented) {
    this.movie = movie;
    this.daysRented = daysRented;
  }

  public Movie getMovie() {
    return movie;
  }

  public int getDaysRented() {
    return daysRented;
  }

  public double getCharge() {
    return movie.getCharge(daysRented);
  }

  public int getPoints() {
    return movie.getPoints(daysRented);
  }

  public String getLineItem() {
    return
      movie.getTitle() + " " + getCharge();
  }
}


public class Movie {

  public enum Type {
    REGULAR, NEW_RELEASE, CHILDREN, UNKNOWN;
  }

  private String title;
  Price price;

  public Movie(
```

```
    String title, Movie.Type priceCode) {
    this.title = title;
    setPriceCode(priceCode);
  }

  public String getTitle() {
    return title;
  }

  private void setPriceCode(
    Movie.Type priceCode) {
    switch (priceCode) {
    case CHILDREN:
      price = new ChildrensPrice();
      break;
    case NEW_RELEASE:
      price = new NewReleasePrice();
      break;
    case REGULAR:
      price = new RegularPrice();
      break;
    default:
      throw new IllegalArgumentException(
        "invalid price code");
    }
  }

  public double getCharge(int daysRented) {
    return price.getCharge(daysRented);
  }

  public int getPoints(int daysRented) {
    return price.getPoints(daysRented);
  }

}
```

```
public abstract class Price {
  abstract double getCharge(int daysRented);

  int getPoints(int daysRented) {
    return 1;
  }
}


public class ChildrensPrice extends Price {
  @Override
  double getCharge(int daysRented) {
    double amount = 1.5;
    if (daysRented > 3)
      amount += (daysRented - 3) * 1.5;
    return amount;
  }
}

public class RegularPrice extends Price {
  @Override
  public double getCharge(int daysRented) {
    double amount = 2;
    if (daysRented > 2)
      amount += (daysRented - 2) * 1.5;
    return amount;
  }
}

public class NewReleasePrice extends Price {
  @Override
  public double getCharge(int daysRented) {
    return daysRented * 3;
  }

  @Override
  int getPoints(int daysRented) {
    if (daysRented > 1)
      return 2;
    return 1;
  }
}
```

Moving Towards Readability

When asked "Why do you test?", Josh Graham gave the following answer:

> To create a tiny universe where the software exists to do one thing and do it well.

The example tests could have been written for many reasons, let's assume the motivators that matter to us are: enable refactoring, immediate feedback, and breaking a problem up into smaller pieces. I would argue that the tests fit well for our first two motivators, but comes up short when it comes to breaking a problem up into smaller pieces. When writing these tests it's obvious and clear where "duplication" lies and how "common" pieces can be pulled into helper methods. Unfortunately, each time we extract a method we're creating an opportunity to introduce complication to our tiny universes. The right abstractions can reduce complexity; however, it's often unclear which abstraction within a test will provide the most value to the team.

DRY has been applied to the tests as it would be to production code. At first glance this may seem like a reasonable approach; however, test code and production code run in drastically different ways. Production code collaborates to provide a single running application, and it's generally wise to avoid duplicating concepts within that application. Tests do not collaborate; it's universally accepted that inter-test dependency is an anti-pattern. If we think of tests as tiny, independent universes, then I would argue characters written in one test should not be considered duplication if they appear in another test as well.

Still, I recognize that pragmatic removal of duplication can add to maintainability. The examples that follow will address issues such as *We've grouped david, john, pat, & steve despite the fact that none of them interact with each other in any way whatsoever* not by duplicating every character, but by introducing local and global patterns that I find superior.

When I think about the current state of the tests, I remember Pat Farley describing some tests as having been *made DRY with a blowtorch.*

Rather than viewing our tests in any form of aggregation, we can shift our focus to viewing each test as a tiny universe; each test can be an individual procedural program that has a single responsibility. If we want to keep our individual procedural programs as tiny universes, we'll likely make many decisions differently.

- We won't test diverse customers at the same time.
- We won't create diverse customers that have nothing to do with each other.
- We won't extract methods for a single string return value.
- We'll create data where we need it, not as part of a special framework method.

In general, I find applying DRY to a subset of tests to be an anti-pattern. Within a single test, DRY can often apply. Likewise, globally appropriate DRY application is often a good choice. However, once

you start applying DRY at a test group level you often increase the complexity of your individual procedures where a local or global solution would have been superior.

For those that enjoy acronyms, when writing tests you should prefer DAMP (Descriptive And Maintainable Procedures) to DRY.

The remainder of the chapter will demonstrate the individual steps we can take to create tests so small they become trivial to immediately understand.

Replace Loop with Individual Tests

The first step in moving to more readable tests is breaking the iteration into individual tests. The following code provides the same regression protection and immediate feedback as the original, while also explicitly giving us more information: passing and failing assertions that may give additional clues as to where the problem exists.

```
public class CustomerTest {
  Customer john, steve, pat, david;
  String johnName = "John",
    steveName = "Steve",
    patName = "Pat",
    davidName = "David";
  Customer[] customers;

  @Before
  public void setup() {
    david = ObjectMother
      .customerWithNoRentals(davidName);
    john = ObjectMother
      .customerWithOneNewRelease(johnName);
    pat = ObjectMother
      .customerWithOneOfEachRentalType(
        patName);
    steve = ObjectMother
      .customerWithOneNewReleaseAndOneRegular(
        steveName);
    customers = new Customer[] {
      david, john, steve, pat };
  }

  @Test
  public void davidStatement() {
    assertEquals(
      expStatement(
        "Rental record for %s\n%sAmount " +
        "owed is %s\nYou earned %s " +
        "frequent renter points",
        david,
        rentalInfo(
          "\t", "", david.getRentals())),
      david.statement());
```

```
  }

  @Test
  public void johnStatement() {
    assertEquals(
      expStatement(
        "Rental record for %s\n%sAmount " +
        "owed is %s\nYou earned %s " +
        "frequent renter points",
        john,
        rentalInfo(
          "\t", "", john.getRentals())),
      john.statement());
  }

  @Test
  public void patStatement() {
    assertEquals(
      expStatement(
        "Rental record for %s\n%sAmount " +
        "owed is %s\nYou earned %s " +
        "frequent renter points",
        pat,
        rentalInfo(
          "\t", "", pat.getRentals())),
      pat.statement());
  }

  @Test
  public void steveStatement() {
    assertEquals(
      expStatement(
        "Rental record for %s\n%s" +
        "Amount owed is %s\nYou earned %s " +
        "frequent renter points",
        steve,
        rentalInfo(
          "\t", "", steve.getRentals())),
      steve.statement());
  }

  public static String rentalInfo(
```

```
    String startsWith,
    String endsWith,
    List<Rental> rentals) {
    String result = "";
    for (Rental rental : rentals)
      result += String.format(
        "%s%s\t%s%s\n",
        startsWith,
        rental.getMovie().getTitle(),
        rental.getCharge(),
        endsWith);
    return result;
  }

  public static String expStatement(
    String formatStr,
    Customer customer,
    String rentalInfo) {
    return String.format(
      formatStr,
      customer.getName(),
      rentalInfo,
      customer.getTotalCharge(),
      customer.getTotalPoints());
  }
}
```

The following output is the result of running the above test.

```
JUnit version 4.11
.E.E..E
There were 3 failures:
1) johnStatement(CustomerTest)
org.junit.ComparisonFailure: expected:<...or John
        Godfather 4[        ]9.0
Amount owed is 9...> but was:<...or John
        Godfather 4[ ]9.0
Amount owed is 9...>
2) steveStatement(CustomerTest)
org.junit.ComparisonFailure: expected:<...r Steve
        Godfather 4[        9.0
        Scarface        ]3.5
```

```
Amount owed is 1...> but was:<...r Steve
        Godfather 4[ 9.0
        Scarface ]3.5
Amount owed is 1...>
3) patStatement(CustomerTest)
org.junit.ComparisonFailure: expected:<...for Pat
        Godfather 4[        9.0
        Scarface        3.5
        Lion King        ]1.5
Amount owed is 1...> but was:<...for Pat
        Godfather 4[ 9.0
        Scarface 3.5
        Lion King ]1.5
Amount owed is 1...>

FAILURES!!!
Tests run: 4,  Failures: 3
```

At this point we have more clues about which tests are failing and where; specifically, we know that `davidStatement` is passing, so the issue must exist in the printing of rental information. Unfortunately, we don't currently have any strings to quickly look at to determine whether the error exists in our expected or actual values.

Expect Literals

The next step in increasing readability is expecting literal values. If you know where the problem exists, having DRY tests can help ensure you type the fewest amount of characters. That said…

> "Programming is not about typing… it's about thinking." –Rich Hickey

At this point, our tests have become much smaller universes, so small that I find myself wondering why I call a parameterized method, once, that does nothing more than return a `String`. Within my tiny universe it would be much easier to simply use a `String` literal.

A few printlns and copy-pastes later, my tests are much more explicit, and my universes have gotten even smaller.

```
public class CustomerTest {
  Customer john, steve, pat, david;
  String johnName = "John",
    steveName = "Steve",
    patName = "Pat",
    davidName = "David";
  Customer[] customers;

  @Before
  public void setup() {
    david = ObjectMother
      .customerWithNoRentals(davidName);
    john = ObjectMother
      .customerWithOneNewRelease(johnName);
    pat = ObjectMother
      .customerWithOneOfEachRentalType(
        patName);
    steve = ObjectMother
      .customerWithOneNewReleaseAndOneRegular(
        steveName);
    customers = new Customer[] {
      david, john, steve, pat };
  }

  @Test
  public void davidStatement() {
    assertEquals(
      "Rental record for David\nAmount " +
```

```
      "owed is 0.0\n" +
      "You earned 0 frequent renter points",
      david.statement());
  }

  @Test
  public void johnStatement() {
    assertEquals(
      "Rental record for John\n\t" +
      "Godfather 4\t9.0\n" +
      "Amount owed is 9.0\n" +
      "You earned 2 frequent renter points",
      john.statement());
  }

  @Test
  public void patStatement() {
    assertEquals(
      "Rental record for Pat\n\t" +
      "Godfather 4\t9.0\n" +
      "\tScarface\t3.5\n" +
      "\tLion King\t1.5\n" +
      "Amount owed is 14.0\n" +
      "You earned 4 frequent renter points",
      pat.statement());
  }

  @Test
  public void steveStatement() {
    assertEquals(
      "Rental record for Steve\n\t" +
      "Godfather 4\t9.0\n" +
      "\tScarface\t3.5\n" +
      "Amount owed is 12.5\n" +
      "You earned 3 frequent renter points",
      steve.statement());
  }
}
```

The failure output is exactly the same, but I'm now able to look at the expected value as a simple constant, and reduce my first question to: is my expected value correct?

> note: Some reviewers were offended by the `Customer.getRentals` public method

> (though others were not). If a method such as `getRentals` is something you'd look to remove from your domain model, then the *Expect Literals* refactoring provides you with at least two benefits: the one you've already seen, and the ability to delete the `getRentals` method entirely. These types of improvements (deleting code is always an improvement, regardless of your preferred OO style) are not uncommon; improving tests often allows you to improve your domain model as well.

There's an entire section dedicated to Expect Literals in Improving Assertions

If this were more than a book example, my next step would likely be adding fine grained tests that verify individual methods of each of the classes that are collaborating with `Customer`. Unfortunately, moving in that direction will require discussion around the benefits of fine grained tests and the trade-offs of using mocks and stubs.

If you're interested in jumping straight to this discussion it can be found in the Types of Tests chapter.

Inline Setup

At this point it should be easy to find the source of the failing test; however, our universes aren't quite DAMP just yet.

Have you ever stopped to ask yourself why we use a design pattern (Template Method) in our tests, when an explicit method call is probably the appropriate choice 99% of the time.

Creating instances of `david`, `john`, `pat`, & `steve` in Setup moves characters out of the individual test methods, but doesn't provide us any other advantage. It also comes with the conceptual overhead of each `Customer` being created, whether or not it's used. By adding a level of indirection we've removed characters from tests, but we've forced ourselves to remember who has what rentals. Removing a setup method almost always reveals an opportunity for a local or global improvement within a universe.

In this case, by removing Setup we're able to further limit the number of variables that require inspection when you first encounter a test. With Setup removed you no longer need to look for a Setup method, and you no longer need to care about the `Customer` instances that are irrelevant to your tiny universe.

```
public class CustomerTest {
  @Test
  public void noRentalsStatement() {
    assertEquals(
      "Rental record for David\nAmount " +
      "owed is 0.0\n" +
      "You earned 0 frequent renter points",
      ObjectMother
      .customerWithNoRentals(
        "David").statement());
  }

  @Test
  public void oneNewReleaseStatement() {
    assertEquals(
      "Rental record for John\n\t" +
      "Godfather 4 9.0\n" +
      "Amount owed is 9.0\n" +
      "You earned 2 frequent renter points",
      ObjectMother
      .customerWithOneNewRelease(
        "John").statement());
  }
```

```
  @Test
  public void allRentalTypesStatement() {
    assertEquals(
      "Rental record for Pat\n\t" +
      "Godfather 4 9.0\n" +
      "\tScarface 3.5\n\tLion King 1.5\n" +
      "Amount owed is 14.0\n" +
      "You earned 4 frequent renter points",
      ObjectMother
      .customerWithOneOfEachRentalType(
        "Pat").statement());
  }

  @Test
  public void
  newReleaseAndRegularStatement() {
    assertEquals(
      "Rental record for Steve\n\t" +
      "Godfather 4 9.0\n" +
      "\tScarface 3.5\n" +
      "Amount owed is 12.5\n" +
      "You earned 3 frequent renter points",
      ObjectMother
      .customerWithOneNewReleaseAndOneRegular(
        "Steve").statement());
  }
}
```

By Inlining Setup we get to delete both the Setup method and the `Customer` fields. Our tests are looking nice and slim, and they require almost no navigation to completely understand. I went ahead an renamed the tests, deleted the unused `customers` field, and inlined the single usage fields.

It's confession time: I don't like test names. Technically they're method names, but they're **never** called explicitly. That alone should make you somewhat suspicious. In my opinion, they're glorified comments that come with all the standard warnings: they often grow out of date, and are often a smell emanating from bad code. Unfortunately, most testing frameworks make test names mandatory, and you *should* spend the time to create helpful test names. While we refactored away from the looping test I lazily named my tests based on the customer; however, I was forced create a more appropriate name as a side effect of performing Inline Setup.

I believe this is another example of how well written tests have side effects that improve associated code. I've personally written testing frameworks that make test names optional, that's how little I care about test names. Still, once I performed Inline Setup, the only reasonable choice was to create a somewhat helpful test name.

Our tests are looking better and better, and I'm feeling motivated to clean up every imperfection. There's still one additional step I would take.

Replace ObjectMother with DataBuilder

ObjectMother is an effective tool when the scenarios are limited and constant, but there's no clear way to handle the situation when you need a slightly different scenario. For example, if you wanted to create a test for the `statement` method on a `Customer` with two New Releases, would you add another `ObjectMother` method or would you call the `addRental` method on an instance returned?

Further complicating the issue: it's often hard to know if you're dealing with objects that you can manipulate to your hearts desire or if the objects returned from an ObjectMother are reused in anyway. For example, if you called `ObjectMother.customerWithTwoNewReleases`, can you change the name on one of the New Release instances, or was the same `Movie` supplied to `addRental` twice? You can't know without looking at the implementation.

At this point it would be natural to delete the `ObjectMother` and simply create your domain model instances using `new`. If the number of calls to `new` within your tests will be limited, that's the pragmatic path. However, as the number of calls to `new` grows so does the risk of needing to do a cascading update. Say you have less than a dozen calls to `new Customer(...)` in your tests and you need to add a constructor argument, updating those dozen or less calls will not severely impact your effectiveness. Conversely, if you have one hundred calls to `new Customer(...)` and you add a constructor argument, you're now forced to update the code in one hundred different places.

A `DataBuilder` is an alternative to a scenario based `ObjectMother` that also addresses the cascading update risk. The following `a` class is a `DataBuilder` that will allow us to build our domain objects that aren't tied to any particular scenario.

(I recommend skimming the following builder, we'll revisit *Test Data Builders* in detail in the TestDataBuilder section of Chapter 6)

```
public class a {
  public static CustomerBuilder customer =
    new CustomerBuilder();
  public static RentalBuilder rental =
    new RentalBuilder();
  public static MovieBuilder movie =
    new MovieBuilder();

  public static class CustomerBuilder {
    Rental[] rentals;
    String name;

    CustomerBuilder() {
      this("Jim", new Rental[0]);
    }
```

```
    CustomerBuilder(
      String name, Rental[] rentals) {
      this.name = name;
      this.rentals = rentals;
    }

    public CustomerBuilder w(
      RentalBuilder... builders) {
      Rental[] rentals =
        new Rental[builders.length];
      for (int i=0; i<builders.length; i++) {
        rentals[i] = builders[i].build();
      }
      return
        new CustomerBuilder(name, rentals);
    }

    public CustomerBuilder w(String name) {
      return
        new CustomerBuilder(name, rentals);
    }

    public Customer build() {
      Customer result = new Customer(name);
      for (Rental rental : rentals) {
        result.addRental(rental);
      }
      return result;
    }
  }

  public static class RentalBuilder {
    final Movie movie;
    final int days;

    RentalBuilder() {
      this(new MovieBuilder().build(), 3);
    }

    RentalBuilder(Movie movie, int days) {
      this.movie = movie;
      this.days = days;
```

```
    }

    public RentalBuilder w(
      MovieBuilder movie) {
      return
        new RentalBuilder(
          movie.build(), days);
    }

    public Rental build() {
      return new Rental(movie, days);
    }
  }

  public static class MovieBuilder {
    final String name;
    final Movie.Type type;

    MovieBuilder() {
      this("Godfather 4",
           Movie.Type.NEW_RELEASE);
    }

    MovieBuilder(
      String name, Movie.Type type) {
      this.name = name;
      this.type = type;
    }

    public MovieBuilder w(Movie.Type type) {
      return new MovieBuilder(name, type);
    }

    public MovieBuilder w(String name) {
      return new MovieBuilder(name, type);
    }

    public Movie build() {
      return new Movie(name, type);
    }
  }
}
```

The `a` class is undeniably longer than an `ObjectMother`; however it's not only more general it also puts the decision in your hands to share or not share an object. Let's look at what our test could look like when utilizing a *Test Data Builder*.

> note: w() is an abbreviation for with().

```
public class CustomerTest {
  @Test
  public void noRentalsStatement() {
    assertEquals(
      "Rental record for David\nAmount " +
      "owed is 0.0\nYou earned 0 frequent " +
      "renter points",
      a.customer.w(
        "David").build().statement());
  }

  @Test
  public void oneNewReleaseStatement() {
    assertEquals(
      "Rental record for John\n\t" +
      "Godfather 4 9.0\nAmount owed is " +
      "9.0\nYou earned 2 frequent renter " +
      "points",
      a.customer.w("John").w(
        a.rental.w(
          a.movie.w(NEW_RELEASE))).build()
      .statement());
  }

  @Test
  public void allRentalTypesStatement() {
    assertEquals(
      "Rental record for Pat\n\t" +
      "Godfather 4 9.0\n\tScarface 3.5\n" +
      "\tLion King 1.5\nAmount owed is " +
      "14.0\nYou earned 4 frequent renter " +
      "points",
      a.customer.w("Pat").w(
        a.rental.w(a.movie.w(NEW_RELEASE)),
        a.rental.w(a.movie.w("Scarface").w(
```

```
                    REGULAR)),
            a.rental.w(a.movie.w("Lion King").w(
                    CHILDREN))).build()
        .statement());
  }

  @Test
  public void
  newReleaseAndRegularStatement() {
    assertEquals(
      "Rental record for Steve\n\t" +
      "Godfather 4 9.0\n\tScarface 3.5\n" +
      "Amount owed is 12.5\nYou earned 3 " +
      "frequent renter points",
      a.customer.w("Steve").w(
        a.rental.w(a.movie.w(NEW_RELEASE)),
        a.rental.w(
          a.movie.w(
            "Scarface").w(REGULAR))).build()
      .statement());
  }
}
```

The above test fails in the same way as all the previous versions of testing `statement`. This version does require us to understand the abstract concept and concrete implementation of a *Test Data Builder*; however, there's no guarantee that you would need to visit the `a` class to understand this test - even the first time you encounter the test. The `a` class is a class used globally to create all domain objects for all tests. With that kind of scope, unless this is your first day on a project, it's not really possible that you wouldn't have encountered the `a` class in the past.

To be more clear, the lone responsibility of a *Test Data Builder* is to create domain objects with default values. Thus, even if you've never seen this test before, without navigating to `a` you'll already know that you're creating a customer with sensible defaults. This is a rare example of an abstraction that, despite adding indirection, also makes the test easier to digest.

With a *Test Data Builder* in place it becomes trivial to add an additional test that verifies the case of 2 New Releases, or any other rental combination that you find to be important.

As I previously mentioned, the choice to use a *Test Data Builder* will likely depend on the number of calls to `new` and your tolerance for cascading update risk. I introduce them here due to their frequent usage throughout the book. In practice I like to use `new` while there are half a dozen or less calls to a constructor.

More information on *Test Data Builders* can be found in Nat Pryce's article on Test Data Builders[4]

[4] http://nat.truemesh.com/archives/000714.html

and by skipping directly to the TestDataBuilder section of Chapter 6.

Comparing the Results

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand. –Martin Fowler, Refactoring.
>
> Applied to Unit Testing: Any fool can write a test that helps them today. Good programmers write tests that help the entire team in the future.

Below you can find both the before and after examples, allowing a quick comparison.

Original

```
public class CustomerTest {
  Customer john, steve, pat, david;
  String johnName = "John",
    steveName = "Steve",
    patName = "Pat",
    davidName = "David";
  Customer[] customers;

  @Before
  public void setup() {
    david = ObjectMother
      .customerWithNoRentals(
        davidName);
    john = ObjectMother
      .customerWithOneNewRelease(
        johnName);
    pat = ObjectMother
      .customerWithOneOfEachRentalType(
        patName);
    steve = ObjectMother
      .customerWithOneNewReleaseAndOneRegular(
        steveName);
    customers =
      new Customer[]
      { david, john, steve, pat};
  }

  @Test
  public void getName() {
    assertEquals(
```

```
      davidName, david.getName());
    assertEquals(
      johnName, john.getName());
    assertEquals(
      steveName, steve.getName());
    assertEquals(
      patName, pat.getName());
  }

  @Test
  public void statement() {
    for (int i=0; i<customers.length; i++) {
      assertEquals(
        expStatement(
          "Rental record for %s\n" +
          "%sAmount owed is %s\n"  +
          "You earned %s frequent " +
          "renter points",
          customers[i],
          rentalInfo(
            "\t", "",
            customers[i].getRentals())),
        customers[i].statement());
    }
  }

  @Test
  public void htmlStatement() {
    for (int i=0; i<customers.length; i++) {
      assertEquals(
        expStatement(
          "<h1>Rental record for " +
          "<em>%s</em></h1>\n%s" +
          "<p>Amount owed is <em>%s</em>" +
          "</p>\n<p>You earned <em>%s" +
          " frequent renter points</em></p>",
          customers[i],
          rentalInfo(
            "<p>", "</p>",
            customers[i].getRentals())),
        customers[i].htmlStatement());
    }
```

```
  }

  @Test
  (expected=IllegalArgumentException.class)
  public void invalidTitle() {
    ObjectMother
      .customerWithNoRentals("Bob")
      .addRental(
        new Rental(
          new Movie("Crazy, Stupid, Love.",
                    Movie.Type.UNKNOWN),
          4));
  }

  public static String rentalInfo(
    String startsWith,
    String endsWith,
    List<Rental> rentals) {
    String result = "";
    for (Rental rental : rentals)
      result += String.format(
        "%s%s\t%s%s\n",
        startsWith,
        rental.getMovie().getTitle(),
        rental.getCharge(),
        endsWith);
    return result;
  }

  public static String expStatement(
    String formatStr,
    Customer customer,
    String rentalInfo) {
    return String.format(
      formatStr,
      customer.getName(),
      rentalInfo,
      customer.getTotalCharge(),
      customer.getTotalPoints());
  }
}
```

Final

```
public class CustomerTest {
  @Test
  public void getName() {
    assertEquals(
      "John",
      a.customer.w(
        "John").build().getName());
  }

  @Test
  public void noRentalsStatement() {
    assertEquals(
      "Rental record for David\nAmount " +
      "owed is 0.0\nYou earned 0 frequent " +
      "renter points",
      a.customer.w(
        "David").build().statement());
  }

  @Test
  public void oneNewReleaseStatement() {
    assertEquals(
      "Rental record for John\n" +
      "\tGodfather 4 9.0\n" +
      "Amount owed is 9.0\n" +
      "You earned 2 frequent renter points",
      a.customer.w("John").w(
        a.rental.w(
          a.movie.w(
            NEW_RELEASE))).build()
      .statement());
  }

  @Test
  public void allRentalTypesStatement() {
    assertEquals(
      "Rental record for Pat\n" +
      "\tGodfather 4 9.0\n" +
      "\tScarface 3.5\n" +
      "\tLion King 1.5\n" +
```

```
      "Amount owed is 14.0\n" +
      "You earned 4 frequent renter points",
      a.customer.w("Pat").w(
        a.rental.w(a.movie.w(NEW_RELEASE)),
        a.rental.w(
          a.movie.w("Scarface").w(REGULAR)),
        a.rental.w(
          a.movie.w("Lion King").w(
            CHILDREN))).build().statement());
  }

  @Test
  public void
  newReleaseAndRegularStatement() {
    assertEquals(
      "Rental record for Steve\n" +
      "\tGodfather 4 9.0\n" +
      "\tScarface 3.5\n" +
      "Amount owed is 12.5\n" +
      "You earned 3 frequent renter points",
      a.customer.w("Steve").w(
        a.rental.w(a.movie.w(NEW_RELEASE)),
        a.rental.w(
          a.movie.w("Scarface").w(
            REGULAR))).build().statement());
  }

  @Test
  public void noRentalsHtmlStatement() {
    assertEquals(
      "<h1>Rental record for <em>David" +
      "</em></h1>\n<p>Amount owed is <em>" +
      "0.0</em></p>\n<p>You earned <em>0 " +
      "frequent renter points</em></p>",
      a.customer.w(
        "David").build().htmlStatement());
  }

  @Test
  public void oneNewReleaseHtmlStatement() {
    assertEquals(
      "<h1>Rental record for <em>John</em>" +
```

```
      "</h1>\n<p>Godfather 4 9.0</p>\n" +
      "<p>Amount owed is <em>9.0</em></p>" +
      "\n<p>You earned <em>2 frequent " +
      "renter points</em></p>",
      a.customer.w("John").w(
        a.rental.w(
          a.movie.w(
            NEW_RELEASE))).build()
      .htmlStatement());
  }

  @Test
  public void allRentalTypesHtmlStatement() {
    assertEquals(
      "<h1>Rental record for <em>Pat</em>" +
      "</h1>\n<p>Godfather 4 9.0</p>\n" +
      "<p>Scarface 3.5</p>\n<p>Lion King" +
      " 1.5</p>\n<p>Amount owed is <em>" +
      "14.0</em></p>\n<p>You earned <em>" +
      "4 frequent renter points</em></p>",
      a.customer.w("Pat").w(
        a.rental.w(a.movie.w(NEW_RELEASE)),
        a.rental.w(
          a.movie.w("Scarface").w(REGULAR)),
        a.rental.w(
          a.movie.w("Lion King").w(
            CHILDREN))).build()
      .htmlStatement());
  }

  @Test
  public void
  newReleaseAndRegularHtmlStatement() {
    assertEquals(
      "<h1>Rental record for <em>Steve" +
      "</em></h1>\n<p>Godfather 4 9.0</p>" +
      "\n<p>Scarface 3.5</p>\n<p>Amount " +
      "owed is <em>12.5</em></p>\n<p>" +
      "You earned <em>3 frequent renter " +
      "points</em></p>",
      a.customer.w("Steve").w(
        a.rental.w(a.movie.w(NEW_RELEASE)),
```

```
        a.rental.w(
          a.movie.w("Scarface").w(
            REGULAR))).build()
      .htmlStatement());
  }

  @Test
  (expected=IllegalArgumentException.class)
  public void invalidTitle() {
    a.customer.w(
      a.rental.w(
        a.movie.w(UNKNOWN))).build();
  }
}
```

Final Thoughts on our Tests

The tests in this chapter are fairly simple, and yet they still provide more than enough content to create discussion among most software engineers.

Whether you prefer the original or final versions of `CustomerTest`, it's undeniable that the final version creates far tinier universes to work within. At this point you should have a fairly deep understanding of this simple example. The fact that you possess that deep understanding can be dangerous to ignore. If you write tests assuming the same level of understanding, you force future maintainers to gain that understanding. Conversely, the tests from the final example put all of the test data either directly in the test or in what should be a globally understood class.

> The decision to write DRY or DAMP tests often comes down to whether or not you want to force future maintainers to deeply understand code written strictly for testing purposes.

An interesting side note: Despite replacing DRY tests with DAMP tests, the overall number of lines in the `CustomerTest` class barely changed.

The 'Final' version of `CustomerTest` improved in a few subtle ways that weren't previously emphasized.

- A test that contains more than one assertion (or one assertion that lives in a loop) will terminate on the first failure. By breaking the iteration into individual tests we were able to see all of the failures generated by our domain code change.
- The `invalidTitle` test uses the same instance creation code that all of the other `Customer` tests use. Now that all `Customer`, `Rental`, and `Movie` instances are created by a `DataBuilder` you can make constructor argument changes and the only consequence will be making a change to the `build` method for the associated `*Builder` class.

If you're with me this far, it should be relatively clear what a DAMP test is, and that I believe them to be far more valuable than DRY tests. From here we'll drop a bit into theory, then straight into deeper examples of how to evolve your tests towards a DAMP style, and finally we'll finish with test suite level improvements and what to avoid once you venture on to *Broad Stack Tests*.

Motivators

There are many ways to succeed while writing tests; however, let's start with an example of the more common path.

Let's imagine you read Unit Testing Tips: Write Maintainable Unit Tests That Will Save You Time And Tears[5] and decide that Roy Osherove has shown you the light. You're going to write all your tests with Roy's suggestions in mind. You get the entire team to read Roy's article and everyone adopts the patterns.

Things are going well until you start accidentally breaking tests that someone else wrote and you can't figure out why. It turns out that some object created in the setup method is causing unexpected failures due to an unexpected side-effect of your 'minor' change. You're frustrated, having been burned by Setup, and you remember the blog entry by Jim Newkirk where he discussed Why you should not use SetUp and TearDown in NUnit[6]. Now you're stuck with a Setup heavy test suite, and a growing suspicion that you've gone down the wrong path.

You do more research on setup and stumble upon Inline Setup. You can entirely relate and go on a mission to switch all the tests to xUnit.net; xUnit.net removes the concept of setup entirely.

Everything looks good initially, but then a few constructors start needing more dependencies. Every test creates it's own instance of an object; you moved the object creation out of the setup and into each individual test. So now every test that creates that object needs to be updated. It becomes painful every time you add an argument to a constructor. You're once again left feeling like you've been burnt by following "expert" advice.

I would argue that the root problem is: you never asked yourself 'why?'. Why are you writing tests in the first place? Each testing practice you've chosen, what motivated you to adopt it?

You wont write better software by **blindly** following advice. This is especially true given that much of the advice around testing is inconsistent or outright conflicting. While I'm writing this chapter there's currently a twitter discussion with Martin Fowler, Michael Feathers, Bob Martin, Prag Dave, and David Heinemeier Hansson (all well respected and successful software engineers) where there are drastically conflicting opinions on how to effectively test. It's safe to say, if there's a universally right way, we haven't found it yet.

> It's worth noting that the articles from Roy and Jim are quite old. Roy has since changed his mind on setup (his current opinions can be found at artofunittesting.com), and I'm sure Jim has updated his opinions as well. The point of the example is to show how it's easy to blindly follow advice that sounds good, not to tie a good or bad idea with an individual.

[5] http://msdn.microsoft.com/en-us/magazine/cc163665.aspx

[6] http://jamesnewkirk.typepad.com/posts/2007/09/why-you-should-.html

Back to our painful journey above: your intentions were good. You want to write better software, so you followed some reasonable advice. Unfortunately, the advice you've chosen to follow left you with more pain than happiness. Your tests aren't providing enough value to justify their effort and if you keep going down this path you'll inevitably conclude that testing is stupid and it should be abandoned.

If you've traveled the path above or if you aren't regularly writing unit tests, you may find yourself wondering why other developers find them so essential. Ultimately I believe the answer boils down to selecting testing patterns based on what's motivating you to test.

The remainder of this chapter will focus on defining testing motivators.

Validating the System

Common motivators that would be a subset of Validating the System

- Immediate Feedback That Things Work as Expected
- Prevent Future Regressions

Static languages like Java provide a compiler that protects you from a certain class of errors; however, unit tests often prove their value when you need to verify not the type of a result, but the value of the result. For example, If your shopping cart cannot correctly calculate the total of each contained item, it won't really matter that the result is an `Integer`.

For this reason, I would argue that every codebase would benefit from, if nothing else, wrapping a few unit tests around the features of the system that if broken would cause the system to become unusable.

Theoretically, you could write a test to validate every feature of your system; however, I believe you would quickly find this task to be substantial and not necessarily worth your time - certain features of your system will likely be more important than others.

There's a common term in finance: ROI

> Return on investment (ROI) is the concept of an investment of some resource yielding a benefit to the investor. A high ROI means the investment gains compare favorably to investment cost.

When I'm motivated to write a test to validate the system, I like to look at the test from an ROI point of view. My favorite example for demonstrating how I choose based on ROI is the following:

Given a system where customers are looked up exclusively by Social Security Number

- I would unit test that a Social Security Number is valid at account creation time
- I would not unit test that a users name is alpha-numeric.

Losing a new account based on an invalid Social Security Number could be rather harmful to a business; however, storing an incorrect name for a limited amount of time should have no impact on successful use of the system.

As long as everyone on the team understands the ROI of the various features, you could trust everyone to make the right call on when and when not to test based on ROI. If your team cannot reasonably grant that responsibility and power to each team member then it will likely make sense to either pair program or err on the side of over testing and evaluating the ROI of each test during a code review.

Tests written to validate the system are often used both to verify that the system currently works as expected as well as to prevent future regression.

Code Coverage

Automated code coverage metrics can be a wonderful thing when used correctly. Upon joining a project I often use code coverage to get a feel for the level of quality that I can expect from the application code. A low coverage percentage can show probable lack of quality - though I would consider it more of a smell than a guarantee. A high coverage percentage would make me feel better about the likelihood of finding a well designed codebase, but that's also more of a smell than a guarantee.

> I expect a high level of coverage. Sometimes managers require one. There's a subtle difference. –Brian Marick

I tend to agree with Martin Fowler's view on the subject: *If you are testing thoughtfully and well, I would expect a coverage percentage in the upper 80s or 90s. I would be suspicious of anything like 100% - it would smell of someone writing tests to make the coverage numbers happy, but not thinking about what they are doing.*

Once upon a time a consultancy went as far as putting "100% code coverage" in their contracts. It was a noble goal; unfortunately, a few years later the same consultancy was presenting on the topic of: How to fail with 100% test coverage. There are various problems with pushing towards 100%:

- You'll have to test language and framework features
- You'll have to find a way to test every private method
- You'll have to maintain a lot of tests with negative ROI.
- …

> I find that code coverage metrics over time may signal an interesting change that you may have otherwise missed, but as an absolute number, it's not very useful. –John Hume

> My favorite "100% code coverage" story involves a team that added a bunch of tests to get to 100%… but didn't add any assertions. Code coverage and verification are not the same thing. –Kent Spillner

I suspect most projects will suffer from the opposite, *not enough coverage*. The good news is it's quite simple to run a coverage tool and determine which pieces of code are untested.

I've had success using EMMA and Clover, and John Hume recently pointed me at Cobertura. Code coverage tools are easy to work with; there's no reason you couldn't try a few and decide which you prefer.

Again, code coverage tools are great. I personally strive for around 80% coverage. If you're looking to get above 80%, it would not surprise me to find tests that have code-coverage as their lone motivator.

Enable Refactoring

Getting test coverage on an untested codebase is always painful; however, it's also essential if you're planning to make any changes within the codebase. With the proper tests in place, you should be able to rewrite the internals of a codebase without breaking any of the existing contracts.

In addition to helping you prevent regression, creating tests can also give you direction on where the application can be logically broken up. While writing tests for a codebase you should keep track of dependencies that need to be instantiated, mocked or stubbed but have nothing to do with the current functionality you are focusing on. In general, these are the pieces that should be broken out into components that are easily stubbed (ideally in 1 or 0 lines).

Document the Behavior of the System

When encountering a codebase for the first time, some developers go straight to the tests. These developers read the tests, test names as well as method bodies, to determine the how and why the system works as it does. These same developers enjoy the benefits of automated tests, but they value the documentation of tests almost as much or more than the functional aspect of the tests.

It's absolutely true that the code doesn't lie, and both correct and incorrect comments (including test names) can often give a view into what a developer was thinking when the test was written. If developers use tests as documentation, it's only natural that they create many tests, some of which would likely be unnecessary if they didn't exist solely to document the system.

Before you go deleting what appear to be superfluous tests, make sure you don't have someone on the team that sees your worthless test as essential documentation.

Your Manager Told You To

If this were your only motivator for writing a test, I think you'd be in a very counterintuitive position. If you write worthless tests you're sure to anger your manager. Given that you're forced to write

"meaningful" tests, I believe you'd want to write the most maintainable tests possible despite your lack of additional motivators. I imagine that you'll want to spend as little time as possible reading and writing tests, and the only way I see accomplishing that is by focusing on maintainability.

Thus, even if you don't particularly value testing, it will likely benefit you to seek out the most maintainable way to write tests in your context.

Test Driven Development

Common motivators that would be a subset of TDD

- Breaking a Problem up into Smaller Pieces
- Defining the "Simplest Thing that Could Possibly Work"
- Improved Design

Unit Testing and TDD are often incorrectly conflated and referred to by either name. Unit testing is an umbrella name for a subset of tests. TDD has a very specific definition:

> Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. –Wikipedia

It's not necessary to write unit tests to TDD, nor is it necessary to TDD to write unit tests.

That said, there's a reason that the terms are often conflated: If you're practicing TDD, then you're very likely also writing a substantial amount of unit tests. A codebase written by developers dogmatically following TDD would theoretically contain no code that wasn't written as a result of a failing test. Proponents of TDD often claim that the results of TDD give the existing team and future maintainers a greater level of confidence.

TDD's development cycle is also very appealing to developers who can find a large problem overwhelming, but are able to quickly break a large problem down into many smaller tests that, when combined, solve the larger problem. Rather than focusing on the single large problem and trying to write code that solves for every variable, the developers will focus on writing tests for each individual variable and growing the code in a way where each test keeps passing and each variable is dealt with individually.

Incredibly large and complicated problems don't seem nearly as daunting when programmers are able to focus exclusively on the task at hand: **make the individual test pass**. In addition, all of the previously written tests provide a safety net, thus allowing you to (harmlessly) ignore all prior constraints.

Proponents of TDD generally believe it promotes superior design as well. The two following reasons are the most often used when describing the design benefits of TDD:

- By focusing on the test cases first, a developer is forced to imagine how the functionality will be used by clients.
- TDD leads to more modularized, flexible, and extensible code by requiring that the developers think of the software in terms of small units that can be written and tested independently and integrated together later.

In my opinion every developer should practice TDD at some point in their career. Utilizing TDD at the right moment will unquestionably make you more productive. That said, the frequency of those moments often depends greatly on the individual. Only through experience can a developer know how to judge whether the current moment would benefit or suffer from switching to a TDD cycle.

An anonymous comment once appeared on my blog:

> The developers that know how to write tests correctly are very rare, and only those developers can really do TDD. The rest end up with a nest of poorly written brittle tests that don't fully test the code.

It's my hope that this book will help increase the number of developers that are productively unit testing. Still, it's perfectly reasonable to delete a test that provided value as part of a TDD cycle, but no longer has positive ROI.

Customer Acceptance

Unit Testing to achieve customer acceptance would be an interesting choice. Rarely would a domain expert be willing to sift through all of the unit tests to determine whether or not they're willing to sign off on the system. Thus, I imagine you'd need to devise some type of filtering that allowed the domain expert to drill down to what they believed to be important.

My default choice is to enable the domain expert to write and maintain tests in a tool designed for high level tests; removing developers and unit tests almost entirely from the acceptance process. However, if the developers must be responsible for writing the tests used for customer acceptance, I would devise a plan to annotate the appropriate unit tests and provide a well formatted report based on the automated results.

In my experience, developers are willing to support customer acceptance low level tests that can quickly be debugged when they fail. Conversely, I've never seen a developer that was happy to maintain tests that are both strictly for the customer and high level (thus hard to debug).

Ping Pong Pair-Programming

From the c2.com Wiki

> here's how Pair Programming works in my team.

1. A writes a new test and sees that it fails.
2. B implements the code needed to pass the test.
3. B writes the next test and sees that it fails.
4. A implements the code needed to pass the test.

And so on.

While the most popular definition obviously describes a TDD approach, there's no reason you couldn't ping-pong writing the test after. If you're already pair programming, the rhythm created by practicing ping-pong may be the only motivator you need for writing a test. I've seen this approach utilized very successfully.

Once a feature is complete it's often worth your time to examine the associated tests. Many of the recently created tests will be valuable as is. Other tests may provide negative ROI as written, but with small tweaks can be made to produce positive ROI. Finally, any tests that were motivated solely by the development process should be considered for deletion.

What Motivates You (or Your Team)

The primary driver for this chapter is to recognize that tests can be written for many different reasons, and assuming that a test is necessary simply because it exists is not always the right decision. It's valuable to recognize which motivators are responsible for a test that you're creating or updating. If you come across a test with no motivators, do everyone a favor and delete the test.

I often write speculative tests that help me get to feature completion, but are unnecessary in the long term. Those are the tests that I look to delete once a feature is complete. They're valuable to me for brainstorming purposes, but aren't well designed for documentation, regression detection, or any other motivator. Once they've served their purpose, I happily kill them off.

Deleting tests that no longer provide value is an important activity; however, deleting tests is an activity that shouldn't be taken lightly. Each test deletion likely requires at least a little collaboration to ensure that (as I previously mentioned) your valueless test isn't someone else's documentation.

More...

buy now: https://leanpub.com/wewut[7]

sign up for the announcements and offers mailing list (free): http://signup.wewut.com[8]

join the discussion group (free): http://group.wewut.com[9]

[7] https://leanpub.com/wewut
[8] http://signup.wewut.com
[9] http://group.wewut.com