The Addison-Wesley Signature Series

A Kent Beck Signature Book

Growing Object-Oriented Software, Guided by Tests

Steve Freeman
Nat Pryce

Library of Congress Cataloging-in-Publication Data:

Freeman, Steve, 1958-
Growing object-oriented software, guided by tests / Steve Freeman and Nat Pryce.
p. cm.
ISBN 978-0-321-50362-6 (pbk. : alk. paper) 1. Object-oriented programming (Computer science) 2. Computer software--Testing. I. Pryce, Nat. II. Title.
QA76.64.F747 2010
005.1'17--dc22

2009035239

ISBN-13: 978–0–321–50362–6
ISBN-10: 0–321–50362–7
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing October 2009

Foreword

Kent Beck

One of the dilemmas posed by the move to shorter and shorter release cycles is how to release more software in less time—and continue releasing indefinitely. A new perspective is necessary to resolve this dilemma. More than a shift in techniques is needed.

Growing Object-Oriented Software, Guided by Tests presents such a new perspective. What if software wasn't "made," like we make a paper airplane—finish folding it and fly it away? What if, instead, we treated software more like a valuable, productive plant, to be nurtured, pruned, harvested, fertilized, and watered? Traditional farmers know how to keep plants productive for decades or even centuries. How would software development be different if we treated our programs the same way?

I am most impressed by how this book presents both the philosophy and mechanics of such a shift in perspective. It is written by practitioners who code—and teach others to code—well. From it you can learn both how to program to sustain productivity and how to look at your programs anew.

The style of test-driven development presented here is different from what I practice. I can't yet articulate the difference, but I have learned from the clear, confident presentation of the authors' techniques. The diversity of dialects has given me a new source of ideas to further refine my own development. *Growing Object-Oriented Software, Guided by Tests*, presents a coherent, consistent system of development, where different techniques support each other.

I invite you to read *Growing Object-Oriented Software, Guided by Tests*, to follow along with the examples, to learn how the authors think about programming and how they program. The experience will enrich your software development style, help you program—and, just as important, see your programs differently.

Preface

What Is This Book About?

This book is a practical guide to the best way we've found to write object-oriented software: *test-driven development (TDD)*. It describes the processes we follow, the design principles we strive for, and the tools we use. It's founded on our decades of experience, working with and learning from some of the best programmers in the world.

Within the book, we address some of the questions and confusions we see coming up on project after project. How do I fit test-driven development into a software project? Where do I start? Why should I write both unit and end-to-end tests? What does it mean for tests to "drive" development? How do I test *difficult feature X*?

This book is also very much about design and the way our approach to design informs our approach to TDD. If there's one thing we've learned, it's that test-driven development works best when taken as a whole. We've seen teams that can do the raw practices (writing and running tests) but struggle with the result because they haven't also adopted the deeper processes that lie behind it.

Why "Growing" Object-Oriented Software?

We used the term "growing" because it gives a sense of how we develop incrementally. We have something working at all times, making sure that the code is always as well-structured as possible and thoroughly tested. Nothing else seems to be as effective at delivering systems that work. As John Gall wrote in [Gall03], "A complex system that works is invariably found to have evolved from a simple system that works."

"Growing" also hints at the biological quality we see in good software, the sense of coherence at every level of structure. It ties into our approach to object

orientation which follows Alan Kay's[1] concept of objects being similar to biological cells that send each other messages.

Why "Guided" by Tests?

We write tests *first* because we find that it helps us write better code. Writing a test first forces us to clarify our intentions, and we don't start the next piece of work until we have an unambiguous description of what it should do. The *process* of writing a test first helps us see when a design is too rigid or unfocused. Then, when we want to follow through and fix a design flaw, the tests give us a safety net of regression coverage.

We use the term "guided" because the technique still requires skill and experience. We found test-driven development to be an effective design support tool—once we'd learned how to develop incrementally and to "listen to the tests." Like any serious design activity, TDD requires understanding and sustained effort to work.

We've seen teams that write tests and code at about the same time (and even teams that write the tests first) where the code is a mess and the tests just raise the cost of maintenance. They'd made a start but hadn't yet learned that the trick, as the title of the book suggests, is to let the tests *guide* development. Use the contents of the tests to stay focused on making progress and feedback from the tests to raise the quality of the system.

What about Mock Objects?

Our original motivation for writing the book was to finally explain the technique of using *mock objects*,[2] which we often see misunderstood. As we got deeper into writing, we realized that our community's discovery and use of mock objects was actually an expression of our approach to writing software; it's part of a larger picture.

In the course of the book, we will show how the mock objects technique works, using the jMock library. More specifically, we'll show where it fits into the TDD process and how it makes sense in the context of object-oriented development.

Who Is This Book For?

We wrote this book for the "informed reader." It's intended for developers with professional experience who probably have at least looked at test-driven

1. Alan Kay was one of the authors of Smalltalk and coined the term "object-oriented."
2. Mock objects are substitute implementations for testing how an object interacts with its neighbors.

development. When writing, we imagined we were explaining techniques to a colleague who hadn't come across them before.

To make room for the deeper material we wanted to cover, we've assumed some knowledge of the basic concepts and tools; there are other books that provide a good introduction to TDD.

Is This a Java Book?

We use the Java programming language throughout because it's common enough that we expect our readers to be able at least to understand the examples. That said, the book is really about a set of *techniques* that are applicable to any object-oriented environment.

If you're not using Java, there are equivalents of testing and mocking libraries we use (JUnit and jMock) in many other languages, including C#, Ruby, Python, Smalltalk, Objective-C, and (impressively) C++. There are even versions for more distant languages such as Scala. There are also other testing and mocking frameworks in Java.

Why Should You Listen to Us?

This book distills our experiences over a couple of decades, including nearly ten years of test-driven development. During that time, we have used TDD in a wide range of projects: large message-oriented enterprise-integration systems with an interactive web front-end backed by multiprocessor compute grids; tiny embedded systems that must run in tens of kilobytes of memory; free games used as advertising for business-critical systems; and back-end middleware and network services to highly interactive graphical desktop applications. In addition, we've written about and taught this material at events and companies across the world.

We've also benefited from the experience of our colleagues in the TDD community based in London. We've spent many hours during and after work having our ideas challenged and honed. We're grateful for the opportunity to work with such lively (and argumentative) colleagues.

What Is in This Book?

The book has six parts:

Part I, "Introduction," is a high-level introduction to test-driven development, mock objects, and object-oriented design within the context of a software development project. We also introduce some of the testing frameworks we use in the rest of the book. Even if you're already familiar with TDD, we stilll recommend reading through Chapters 1 and 2 since they describe our approach to software development. If you're familiar with JUnit and jMock, you might want to skip the rest of the introduction.

Part II, "The Process of Test-Driven Development," describes the process of TDD, showing how to get started and how to keep development moving. We dig into the relationship between our test-driven approach and object-oriented programming, showing how the principles of the two techniques support each other. Finally, we discuss how to work with external code. This part describes the concepts, the next part puts them to work.

Part III, "A Worked Example," is an extended example that gives a flavor of how we develop an object-oriented application in a test-driven manner. Along the way, we discuss the trade-offs and motivations for the decisions we take. We've made this quite a long example, because we want to show how some features of TDD become more significant as the code starts to scale up.

Part IV, "Sustainable Test-Driven Development," describes some practices that keep a system maintainable. We're very careful these days about keeping a codebase clean and expressive, because we've learned over the years the costs of letting things slip. This part describes some of the practices we've adopted and explains why we do them.

Part V, "Advanced Topics," looks at areas where TDD is more difficult: complex test data, persistence, and concurrency. We show how we deal with these issues and how this affects the design of the code and tests.

Finally, the appendices include some supporting material on jMock and Hamcrest.

What Is Not in This Book?

This is a technical book. We've left out all the other topics that make a project succeed, such as team organization, requirements management, and product design. Adopting an incremental test-driven approach to development obviously has a close relationship with how a project is run. TDD enables some new activities, such as frequent delivery, and it can be crippled by organizational circumstances, such as an early design freeze or team stakeholders that don't communicate. Again, there are plenty of other books to cover these topics.

Chapter 2

Test-Driven Development with Objects

Music is the space between the notes.

—Claude Debussy

A Web of Objects

Object-oriented design focuses more on the communication between objects than on the objects themselves. As Alan Kay [Kay98] wrote:

> The big idea is "messaging" [...] The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

An object communicates by messages: It receives messages from other objects and reacts by sending messages to other objects as well as, perhaps, returning a value or exception to the original sender. An object has a *method* of handling every type of message that it understands and, in most cases, encapsulates some internal state that it uses to coordinate its communication with other objects.

An object-oriented system is a web of collaborating objects. A system is built by creating objects and plugging them together so that they can send messages to one another. The behavior of the system is an emergent property of the composition of the objects—the choice of objects and how they are connected (Figure 2.1).

This lets us change the behavior of the system by changing the composition of its objects—adding and removing instances, plugging different combinations together—rather than writing procedural code. The code we write to manage this composition is a *declarative* definition of the how the web of objects will behave. It's easier to change the system's behavior because we can focus on *what* we want it to do, not *how*.

Values and Objects

When designing a system, it's important to distinguish between *values* that model unchanging quantities or measurements, and *objects* that have an identity, might change state over time, and model *computational processes*. In the

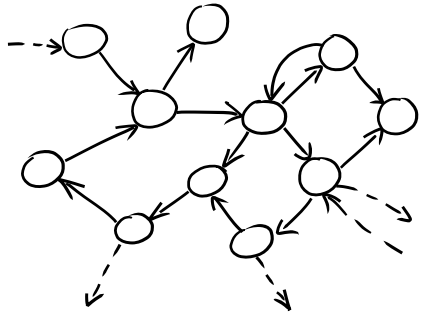

Figure 2.1 *A web of objects*

object-oriented languages that most of us use, the confusion is that both concepts are implemented by the same language construct: classes.

Values are immutable instances that model fixed quantities. They have no individual identity, so two value instances are effectively the same if they have the same state. This means that it makes no sense to compare the identity of two values; doing so can cause some subtle bugs—think of the different ways of comparing two copies of `new Integer(999)`. That's why we're taught to use `string1.equals(string2)` in Java rather than `string1 == string2`.

Objects, on the other hand, use mutable state to model their behavior over time. Two objects of the same type have separate identities even if they have exactly the same state now, because their states can diverge if they receive different messages in the future.

In practice, this means that we split our system into two "worlds": values, which are treated functionally, and objects, which implement the stateful behavior of the system. In Part III, you'll see how our coding style varies depending on which world we're working in.

In this book, we will use the term *object* to refer only to instances with identity, state, and processing—not values. There doesn't appear to be another accepted term that isn't overloaded with other meanings (such as *entity* and *process*).

Follow the Messages

We can benefit from this high-level, declarative approach only if our objects are designed to be easily pluggable. In practice, this means that they follow common *communication patterns* and that the dependencies between them are made explicit. A communication pattern is a set of rules that govern how a group of objects talk to each other: the roles they play, what messages they can send and when, and so on. In languages like Java, we identify object roles with (abstract) interfaces, rather than (concrete) classes—although interfaces don't define everything we need to say.

In our view, *the domain model is in these communication patterns*, because they are what gives meaning to the universe of possible relationships between the objects. Thinking of a system in terms of its dynamic, communication structure is a significant mental shift from the static classification that most of us learn when being introduced to objects. The domain model isn't even obviously visible because the communication patterns are not explicitly represented in the programming languages we get to work with. We hope to show, in this book, how tests and mock objects help us see the communication between our objects more clearly.

Here's a small example of how focusing on the communication between objects guides design.

In a video game, the objects in play might include: *actors*, such as the player and the enemies; *scenery*, which the player flies over; *obstacles*, which the player can crash into; and *effects*, such as explosions and smoke. There are also *scripts* spawning objects behind the scenes as the game progresses.

This is a good classification of the game objects from the players' point of view because it supports the decisions they need to make when playing the game—when interacting with the game from *outside*. This is not, however, a useful classification for the implementers of the game. The game engine has to display objects that are *visible*, tell objects that are *animated* about the passing of time, detect collisions between objects that are *physical*, and delegate decisions about what to do when physical objects collide to *collision resolvers*.

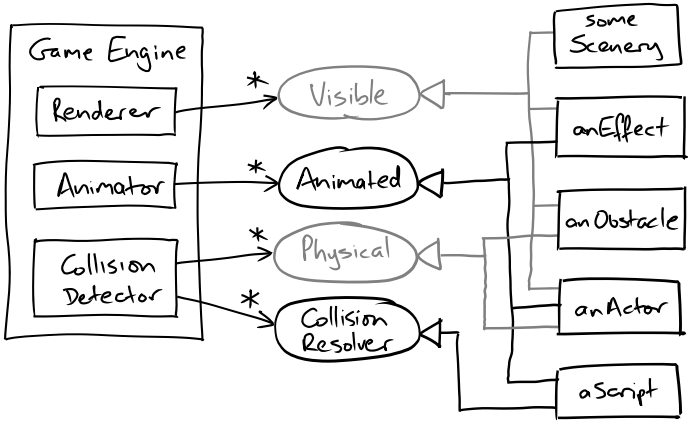

Figure 2.2 *Roles and objects in a video game*

As you can see in Figure 2.2, the two views, one from the game engine and one from the implementation of the in-play objects, are not the same. An Obstacle, for example, is *Visible* and *Physical*, while a Script is a *Collision Resolver* and *Animated* but not *Visible*. The objects in the game play different roles depending

on what the engine needs from them at the time. This mismatch between static classification and dynamic communication means that we're unlikely to come up with a tidy class hierarchy for the game objects that will also suit the needs of the engine.

At best, a class hierarchy represents one dimension of an application, providing a mechanism for sharing implementation details between objects; for example, we might have a base class to implement the common features of frame-based animation. At worst, we've seen too many codebases (including our own) that suffer complexity and duplication from using one mechanism to represent multiple concepts.

Roles, Responsibilities, Collaborators

We try to think about objects in terms of roles, responsibilities, and collaborators, as best described by Wirfs-Brock and McKean in [Wirfs-Brock03]. An object is an implementation of one or more *roles*; a role is a set of related *responsibilities*; and a responsibility is an obligation to perform a task or know information. A *collaboration* is an interaction of objects or roles (or both).

Sometimes we step away from the keyboard and use an informal design technique that Wirfs-Brock and McKean describe, called *CRC cards* (Candidates, Responsibilities, Collaborators). The idea is to use low-tech index cards to explore the potential object structure of an application, or a part of it. These index cards allow us to experiment with structure without getting stuck in detail or becoming too attached to an early solution.

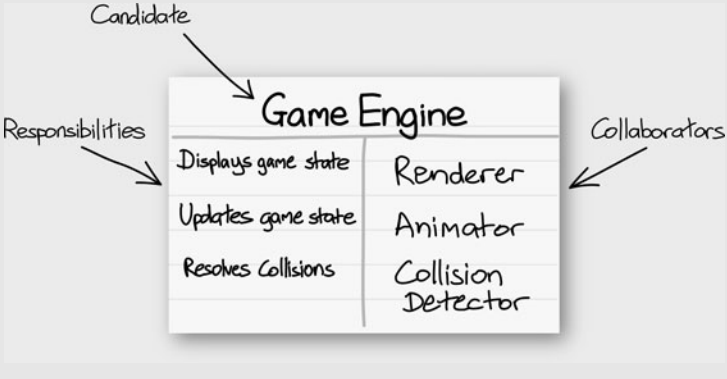

Figure 2.3 *CRC card for a video game*

Tell, Don't Ask

We have objects sending each other messages, so what do they say? Our experience is that the calling object should describe what it wants *in terms of the role* that its neighbor plays, and let the called object decide how to make that happen. This is commonly known as the "*Tell, Don't Ask*" style or, more formally, the *Law of Demeter*. Objects make their decisions based only on the information they hold internally or that which came with the triggering message; they avoid navigating to other objects to make things happen. Followed consistently, this style produces more flexible code because it's easy to swap objects that play the same role. The caller sees nothing of their internal structure or the structure of the rest of the system behind the role interface.

When we don't follow the style, we can end up with what's known as "*train wreck*" code, where a series of getters is chained together like the carriages in a train. Here's one case we found on the Internet:

```
((EditSaveCustomizer) master.getModelisable()
  .getDockablePanel()
    .getCustomizer())
      .getSaveItem().setEnabled(Boolean.FALSE.booleanValue());
```

After some head scratching, we realized what this fragment was *meant* to say:

```
master.allowSavingOfCustomisations();
```

This wraps all that implementation detail up behind a single call. The client of `master` no longer needs to know anything about the types in the chain. We've reduced the risk that a design change might cause ripples in remote parts of the codebase.

As well as hiding information, there's a more subtle benefit from "Tell, Don't Ask." It forces us to make explicit and so name the interactions between objects, rather than leaving them implicit in the chain of getters. The shorter version above is much clearer about *what* it's for, not just *how* it happens to be implemented.

But Sometimes Ask

Of course we don't "tell" everything;[1] we "ask" when getting information from values and collections, or when using a factory to create new objects. Occasionally, we also ask objects about their state when searching or filtering, but we still want to maintain expressiveness and avoid "train wrecks."

For example (to continue with the metaphor), if we naively wanted to spread reserved seats out across the whole of a train, we might start with something like:

1. Although that's an interesting exercise to try, to stretch your technique.

```
public class Train {
  private final List<Carriage> carriages […]
  private int percentReservedBarrier = 70;

  public void reserveSeats(ReservationRequest request) {
    for (Carriage carriage : carriages) {
      if (carriage.getSeats().getPercentReserved() < percentReservedBarrier) {
        request.reserveSeatsIn(carriage);
        return;
      }
    }
    request.cannotFindSeats();
  }
}
```

We shouldn't expose the internal structure of `Carriage` to implement this, not least because there may be different types of carriages within a train. Instead, we should ask the question we really want answered, instead of asking for the information to help us figure out the answer ourselves:

```
public void reserveSeats(ReservationRequest request) {
  for (Carriage carriage : carriages) {
    if (carriage.hasSeatsAvailableWithin(percentReservedBarrier)) {
      request.reserveSeatsIn(carriage);
      return;
    }
  }
  request.cannotFindSeats();
}
```

Adding a query method moves the behavior to the most appropriate object, gives it an explanatory name, and makes it easier to test.

We try to be sparing with queries on objects (as opposed to values) because they can allow information to "leak" out of the object, making the system a little bit more rigid. At a minimum, we make a point of writing queries that describe the intention of the calling object, not just the implementation.

Unit-Testing the Collaborating Objects

We appear to have painted ourselves into a corner. We're insisting on focused objects that send commands to each other and don't expose any way to query their state, so it looks like we have nothing available to assert in a unit test. For example, in Figure 2.4, the circled object will send messages to one or more of its three neighbors when invoked. How can we test that it does so correctly without exposing any of its internal state?

One option is to replace the target object's neighbors in a test with substitutes, or *mock objects*, as in Figure 2.5. We can specify how we expect the target object to communicate with its mock neighbors for a triggering event; we call these specifications *expectations*. During the test, the mock objects assert that they

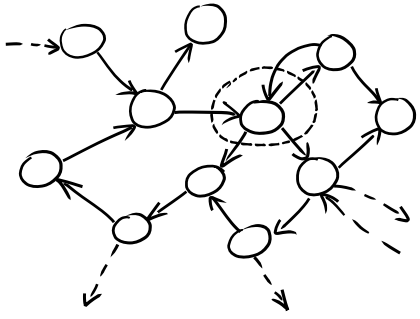

Figure 2.4 *Unit-testing an object in isolation*

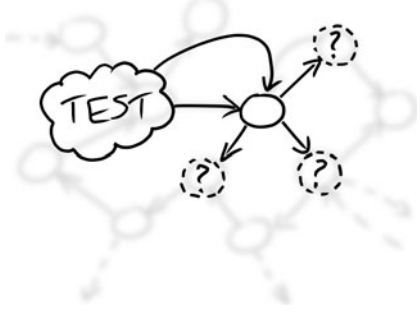

Figure 2.5 *Testing an object with mock objects*

have been called as expected; they also implement any stubbed behavior needed to make the rest of the test work.

With this infrastructure in place, we can change the way we approach TDD. Figure 2.5 implies that we're just trying to test the target object and that we already know what its neighbors look like. In practice, however, those collaborators don't need to exist when we're writing a unit test. We can use the test to help us tease out the supporting roles our object needs, defined as Java interfaces, and fill in real implementations as we develop the rest of the system. We call this *interface discovery*; you'll see an example when we extract an `AuctionEventListener` in Chapter 12.

Support for TDD with Mock Objects

To support this style of test-driven programming, we need to create mock instances of the neighboring objects, define expectations on how they're called and then check them, and implement any stub behavior we need to get through the test. In practice, the runtime structure of a test with mock objects usually looks like Figure 2.6.

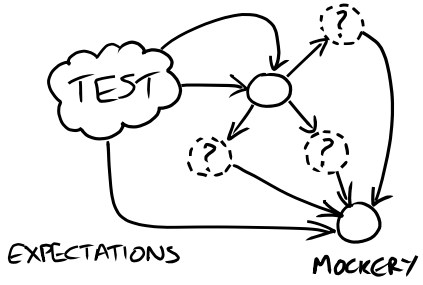

Figure 2.6 *Testing an object with mock objects*

We use the term *mockery*[2] for the object that holds the context of a test, creates mock objects, and manages expectations and stubbing for the test. We'll show the practice throughout Part III, so we'll just touch on the basics here. The essential structure of a test is:

- Create any required mock objects.
- Create any real objects, including the target object.
- Specify how you *expect* the mock objects to be called by the target object.
- Call the triggering method(s) on the target object.
- Assert that any resulting values are valid and that all the expected calls have been made.

The unit test makes explicit the relationship between the target object and its environment. It creates all the objects in the cluster and makes assertions about the interactions between the target object and its collaborators. We can code this infrastructure by hand or, these days, use one of the multiple mock object frameworks that are available in many languages. The important point, as we stress repeatedly throughout this book, is to make clear the intention of every test, distinguishing between the tested functionality, the supporting infrastructure, and the object structure.

2. This is a pun by Ivan Moore that we adopted in a fit of whimsy.

Index

F

G

H

I

J

L

M

N