Extracted from:

Pragmatic Unit Testing

in C# with NUnit, Second Edition

This PDF file contains pages extracted from Pragmatic Unit Testing, one of the Pragmatic Starter Kit series of books for project teams. For more information, visit http://www.pragmaticprogrammer.com/starter_kit.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2006 The Pragmatic Programmers, LLC. All rights reserved.
No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Chapter 6

Using Mock Objects

The objective of unit testing is to exercise just one behavior at a time, but what happens when the method containing that behavior depends on other things—hard-to-control things such as the network, a database, or even specialized hardware?

What if our code depends on other parts of the system—maybe even *many* other parts of the system? If we're not careful, we might find ourselves writing tests that end up (directly or indirectly) initializing nearly every system component just to give the tests enough context to run. Not only is this time-consuming, but it also introduces a ridiculous amount of coupling into the testing process: someone changes an interface or a database table, and suddenly the setup code for our poor little unit test dies mysteriously. With this kind of coupling, sometimes simply adding a new test can cause other tests to fail. Even the best-intentioned developers will become discouraged after this happens a few times, and they eventually may abandon all testing. But there are techniques we can use to help.

In movie and television production, crews will often use *stand-ins* or *doubles* for the real actors. In particular, while the crews are setting up the lights and camera angles, they'll use *lighting doubles*—inexpensive, unimportant people who are about the same height and complexion as the expensive, important actors lounging safely in their luxurious trailers.

The crew then tests their setup with the lighting doubles, measuring the distance from the camera to the stand-in's nose and adjusting the lighting until there are no unwanted shadows, and so on. The obedient stand-in just stands there and doesn't whine or complain about "lacking motivation" for their character in this scene.

What we're going to do in unit testing is similar to using lighting doubles in movies. Instead of testing against the real code itself, we'll use a cheap stand-in that is kind of close to the real code, at least superficially, but will be easier to work with for our nefarious unit testing purposes.

Fortunately, there's a testing pattern that can help: *mock objects*. A mock object is simply a testing replacement for a real-world object. A number of situations can come up where mock objects can help us. Tim Mackinnon [MFC01] offers the following list:

- The real object has nondeterministic behavior (it produces unpredictable results, like a stock market quote feed or random number generator).
- The real object is difficult to set up, requiring a certain file system, database, or network environment.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to confirm that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

Using mock objects, we can get around all of these problems. The three key steps to using mock objects for testing are as follows:

1. Use an interface to describe the relevant methods on the object.

CLICK HERE to purchase this book now.

2. Implement the interface for production code.
3. Implement the interface in a mock object for testing.

The code under test refers to an object only by its interface or base class, so it can remain blissfully ignorant about whether it is using the real object or the mock. Sometimes there's a simpler solution to getting on with our testing, so let's explore that first.

6.1 Stubs

What we need to do is stub out all those uncooperative parts of the real world and replace them with more complicit allies—our own version of lighting doubles. For example, stubs allow us to fake our interaction with a database or the file system.

In many cases, stubs just implement an interface and return dummy values for the methods in said interface. Note that although we can also extract an abstract class, interfaces are preferred since they have no implementation details and therefore provide the loosest coupling.[1] In even simpler cases, all the implemented methods in the stub just throw a `NotImplementedException`.[2]

A common scenario is when there is a class that encapsulates database access[3] but we don't want to configure and populate a real-world database to run simple tests that operate only on data.

```
public class MySqlCustomerRepository
{
  public string[] FindById(long id)
  {
    xxxx xx xxxxx
  }
}
```

---

[1]See [Pug06] and [CA05] for more details on designing with interfaces.

[2]Most IDEs will fill in this exception for us when told to automatically implement the methods for an interface.

[3]The Repository design pattern documented in [Fow03] is one way to accomplish this.

CLICK HERE to purchase this book now.

First, we extract an interface for the methods we need to stub and apply that interface to the class we want to mock:

```
public interface CustomerRepository
{
  string[] FindById(long id);
}
public class MySqlCustomerRepository : CustomerRepository
{
  public string[] FindById(long id)
  {
    xxxx xx xxxxx
  }
}
```

Next, we can return the dummy value that we think will evoke the behavior we want from the `ProductAdoptionService`:

```
public class StubCustomerRepository : CustomerRepository
{
  public string[] FindById(long id)
  {
    return null;
  }
}
```

For now we put the code for this stub class in the same file as the test fixture class that will be using it. Later we'll move it to a more general area where other test fixture classes can access it. Let's plug in the stub to our unit test like so:

```
namespace WebCRM.Test.ProductAdoptionTest
{
  [TestFixture]
  public class NoDataFixture
  {
    [Test]
    public void OverallRateIsZero()
    {
      CustomerRepository customerRepository =
        new StubCustomerRepository();
      ProductAdoptionService service =
        new ProductAdoptionService(customerRepository);
      Assert.That(service.GetPercentage(), Is.EqualTo(0));
    }
  }
}
```

Oops, this won't actually compile. We get an error similar to this:

```
Argument 1: Cannot convert from
 'CustomerRepository' to 'MySqlCustomerRepository'
```

CLICK HERE to purchase this book now.

The `ProductAdoptionService` is still expecting an instance of the concrete `MySqlCustomerRepository` class as the parameter to its constructor. We'll need to change that parameter to accept an object that implements the `CustomerRepository` interface instead. We'll also need to change the field in the `ProductAdoptionService`:

```
public class ProductAdoptionService
{
  CustomerRepository repository;
  public ProductAdoptionService(CustomerRepository repository)
  {
    this.repository = repository;
  }
  xxxxx xx xxxxx xx
}
```

If we're lucky, our test might now pass, and we didn't have to touch a database. In fact, this test could have been written before there was a schema design, database vendor debate, or anything else. By programming to interfaces, we can plug in what we need without being blocked on politicking or other noncoding activities that can slow down a project. Note that we not only get to verify the code being tested produces the results that we want, but we also get to verify that it *interacts* with the stubbed class in the way we expect.

6.2 Fakes

Sometimes you need to do more than return a single, static dummy value to get at the code you're trying to test. Fakes, sometimes known as *static* or *hand-rolled mocks*, improve on stubs by allowing for several different values to be returned. What if you have files on the file system that conform to a certain format and you want to test that you're parsing them correctly?

```
public class DumpFileParser
{
  FileStream stream;
  public DumpFileParser(string fileName)
  {
    stream = File.Open(fileName);
  }
  xxxx xxx xxxx
}
```

The previous code requires a real file on the file system in order to be tested. This can put an unnecessary file system layout burden on the person running the tests, and the disk I/O will slow down the tests.[4] What can you do in a situation like this to make it easier to test? In this case, the class actually discards the supplied filename after the constructor and just operates on the resulting stream.

We'll look at a suboptimal way of making it more testable and then at a more optimal way. It's good to understand what the evils in the world are so that we don't accidentally end up evoking any of them.

What if we used `#define` to tell the code when we were testing? Then it wouldn't use the file system.

```
public class DumpFileParser
{
  FileStream stream;
  public DumpFileParser(string fileName)
  {
#if TESTING
  stream = new MemoryStream();
#else
  stream = File.Open(fileName);
#endif
  }
  xxxx xxx xxxx
}
```

`MemoryStream` is a nifty class in the .NET class library that allows us to make, as you may have guessed, an in-memory stream. Now we have a real `Stream`-derived object that the class can interact with, and it doesn't touch the file system. Before we get too far ahead of ourselves, though, realize that an empty stream has limitations. First, an empty stream doesn't really help you if the code needs to read data from that stream. Many of the tests you write will probably want to supply different data via the stream to make sure the parser behaves correctly. We could figure out various ways to get some test data into place in this scenario, but this approach works around that the code wants the stream to be parameterized; our attempt to test this code has illuminated this.

[4] This doesn't seem like a big deal, but little slowdowns like this add up quickly.

CLICK HERE to purchase this book now.

Also, #if statements strewn throughout the code for testing purposes are difficult to maintain. And, in our opinion, they're ugly to boot.

It might also be tempting to just add an empty constructor to eliminate the need for any of this deep thinking. Although this would "work" in a very narrow sense, there's a good reason there wasn't an empty constructor in the first place: without the Stream being created, the object isn't in a valid state. In this case, invalid state means a probable NullReferenceException whenever you try to do anything with the object. Objects being in a valid state after construction is a core object-oriented design principle, and ignoring it is not the right thing to do in this case. Tests can help drive improvements to the code's design, but this particular example isn't one of them.

Now that we've discussed what won't work, what will work? What if we shifted the responsibility of actually getting the FileStream to the consumers of this class and those consumers passed in a FileStream to the constructor instead of a filename? Doing this transformation would resolve the design feedback we're getting from testing this in the first place:

```
public class DumpFileParser
{
  Stream stream;
  public DumpFileParser(Stream dumpStream)
  {
    this.stream = dumpStream;
  }
}
```

This isn't bad at all. Now the consumers of the class, including the tests, could perform the File.Open() and pass in a FileStream. It may seem like we're just moving the problem around, but we needed to make our code a little shy; specifically, we needed to make it more liberal in what it will accept without complaint.[5] In this case, we aren't using any methods specific to FileStream, so the constructor can actually accept the base class, Stream, instead.[6]

[5] See [HT00] for details on why and how to make code "shy."

[6] You generally want to use interfaces instead of abstract classes for parameters, but Stream doesn't have a base interface [CA05].

CLICK HERE to purchase this book now.

What does that get us? Well, in our tests we can now use the spiffy `MemoryStream` class, like so:

```
[TestFixture]
public class DumpFileParserTest
{
  private StreamWriter writer;
  private DumpFileParser parser;
  private MemoryStream stream;

  [SetUp]
  public void SetUp()
  {
    stream = new MemoryStream();
    writer = new StreamWriter(stream);
  }

  [Test]
  public void EmptyLine()
  {
    writer.WriteLine(string.Empty);
    parser = new Parser(stream);
    Assert.That(xxxx, xxxx);
  }
}
```

Presto! We now have an instant pseudo text file that we can also use to write binary data. Since this operates in memory, we won't incur the performance penalty of disk I/O. Note that this technique works just as well with sockets and other stream-based I/O. Now we can do the testing we need, quickly and conveniently. A nice side effect is that our code is more loosely coupled, yielding a more flexible design that is easier to reuse. One could say that changing the parameter to a `Stream` was a change strictly for the sake of testing, and that observation would be somewhat correct. The other side of the story is that by not programming against a concrete implementation, the code now has a more flexible design. We were led to this by refactoring a very little bit to make things easier to test. This kind of design feedback is the real magic of unit testing, but this is only one simple example.

Faking Collaborators

The `DumpFileParser` class we were just working on does some pretty complicated collation of the data in the stream. If another class depends on `DumpFileParser`, we don't want to make the entire fake stream necessary for it to produce

CLICK HERE to purchase this book now.

the data against which we're trying to test our other class. Besides that it would be really tedious, it adds a whole new dimension of coupling and maintenance to the test code. If we use a real DumpFileParser while testing a collaborating class, we're increasing the work we have to do if DumpFileParser changes or gets removed.

That doesn't sound very pragmatic, so how do we decouple DumpFileParser from the tests of a class that requires a DumpFileParser? It's actually similar to our initial example—we need to abstract things up a level, and then we can supply a variation on DumpFileParser that returns whatever dummy values we need for the purpose of testing the other object. This is known in some circles as creating a *fake* and in other circles as a *static mock*. Let's look at some code:

```
public class Analyzer
{
  private DumpFileParser parser;
  private List<string> reportItems;
  public Analyzer(DumpFileParser parser)
  {
    this.parser = parser;
  }
  public bool ExpectationsMet
  {
    get
    {
      return
        parser.ReportItems.Count == reportItems.Count;
    }
  }
  public byte[] GetNextInstruction()
  {
    xxxxxxxxx
  }
}
```

If we wanted to test the ExpectationsMet property, the ReportItems property on parser would need to be under our control so we can make it return what we want. One way would be to make the ReportItems property on DumpFileParser virtual. We could then subclass and override it for our testing purposes and pass an instance of said subclass into the constructor for Analyzer.

CLICK HERE to purchase this book now.

Although that would work, there's a better way that yields a more flexible, and interface-oriented, design: extract an `interface` called `Parsable` that contains, for the time being, a declaration for the `ReportItems` property getter:

```
public interface Parsable
{
  List<string> ReportItems
  {
    get;
  }
}
```

Then, we can make `DumpFileParser` implement the `Parsable` interface. Next, we change the `Analyzer` constructor's parameter from `DumpFileParser` to `Parsable`. Last, we change the `parser` field in `Analyzer` from the `DumpFileParser` concrete class to be the `Parsable` interface that `DumpFileParser` now implements. When we try to compile, the compiler might tell us that we're using some methods not defined on the `Parsable` interface. We'll need to add those methods to the interface as well:

```
public DumpFileParser : Parsable
{
  xxxxxxx
}
public class Analyzer
{
  private Parsable parser;
  private List<string> reportItems;
  public Analyzer(Parsable parser)
  {
    this.parser = parser;
  }
  xxxxxxxx
}
```

None of the existing consumers of `Analyzer` has to change, and yet, we have just made `Analyzer` easier to test *and* reuse. If we wanted to add the ability to parse another file format, `Analyzer` itself wouldn't have to change to accommodate the extra functionality—only the consumers would by passing in a new class that implemented the `Parsable` interface.

CLICK HERE to purchase this book now.

This is a good example of the advantage of interface-based design, but the point worth mentioning again is that we arrived at this better design by refactoring toward testability. Besides being more testable and reusable, it also means we don't need to wait for another set of programmers to finish implementing the concrete class that our class might be collaborating with. We can fully unit test our class by faking the collaborator's interface, which generally makes integrating with the concrete classes developed by others (or even our future selves) significantly less painful.[7]

Fakes are great, especially when they're simple, but it's also easy to outgrow them, such as when we need to do more than return a single value, for instance. At some point, we want to return values in a certain order each time a method is called. To accomplish this with a fake, we would need to track a `Stack` of return values for a given method:

```
public class FakeParser : Parsable
{
  private Stack<byte[]> bytesToReturn;
  public Stack<byte[]> BytesToReturn
  {
    get { return bytesToReturn; }
    set { bytesToReturn = value; }
  }
  public Boolean ExpectationsMet
  {
    get { return false; }
  }
  public Byte[] GetNextInstruction()
  {
    return BytesToReturn.Pop();
  }
}
```

Although this would work and is a clever way to make a programmable fake, we risk repeating ourselves because we would end up doing this for most methods on our fake. It also gets a little hairier when we have to make them throw specific exceptions at certain points to test failure modes. Surely, there must be a better way.

[7] In many cases, the usually pandemonious step of integration just works.

CLICK HERE to purchase this book now.

6.3 Mock Objects

Sometimes we'll need to test something that uses an existing interface when there are no prewritten stubs or fakes lying around. Often, we can just jump right on in and create a new fake.

But what if the interface that we're mocking is enormous, with dozens of methods and accessors? That could mean a lot of work producing a fake that implements the interface. This is particularly galling if we need only one or two methods from the interface to run our tests, and we can't refactor to break up the interface for some reason.

This is where dynamic mock objects come in. They let us create an object that responds as if it implemented a full interface, but in reality it is totally generic. You need to tell this object only how to respond to the method calls that our code uses. This can represent a considerable savings in time. It'll also give you less code to maintain in the future.

Dynamic mocks are great, but they also make it easy to work around design issues rather than refactoring to fix them. With fakes, because they are hand-rolled, kinks in the design of the code we're trying to test are more obvious.

Some people prefer using hand-rolled fakes and stubs whenever possible so they can get design feedback more directly. Do whatever you are most comfortable with, but pay close attention to what the code is trying to tell you.

The dynamic mock packages operate by creating *proxy objects* that implement the mocked interface at runtime in the underlying implementation. These are objects that are designed to stand in for their real-world counterparts. In the dynamic mock object context, this means we can use a proxy in place of a real object in our tests.

However, we still need to be able to control this generated proxy object—we need to be able to tell it how to respond. This is where the controller comes in.

Click Here to purchase this book now.

A Pragmatic Career

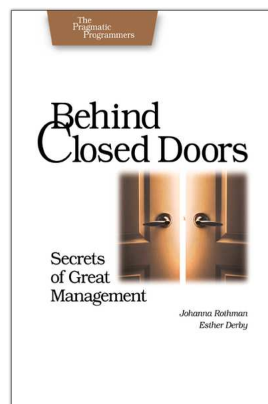
Welcome to the Pragmatic Community. We hope you've enjoyed this title.

If you've enjoyed this book by Johanna Rothman, and want to advance your management career, you'll be interested in seeing what happens *Behind Closed Doors*. And see how you can lead you team to success by using *Agile Retrospectives*.

Behind Closed Doors

You can learn to be a better manager—even a great manager—with this guide. You'll find powerful tips covering:

• Delegating effectively • Using feedback and goal-setting • Developing influence • Handling one-on-one meetings • Coaching and mentoring • Deciding what work to do-and what not to do • . . . and more!

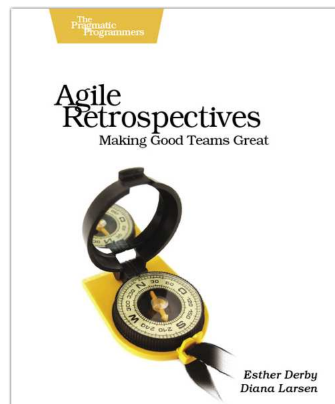


Behind Closed Doors Secrets of Great Management
Johanna Rothman and Esther Derby
(192 pages) ISBN: 0-9766940-2-6. $24.95
http://pragmaticprogrammer.com/titles/rdbcd

Agile Retrospectives

Mine the experience of your software development team continually throughout the life of the project. Rather than waiting until the end of the project—as with a traditional retrospective, when it's too late to help—agile retrospectives help you adjust to change *today*.

The tools and recipes in this book will help you uncover and solve hidden (and not-so-hidden) problems with your technology, your methodology, and those difficult "people issues" on your team.



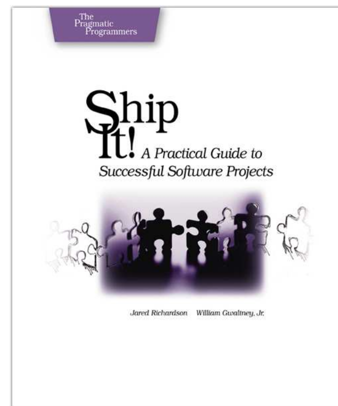
Agile Retrospectives: Making Good Teams Great
Esther Derby and Diana Larsen
(170 pages) ISBN: 0-9776166-4-9. $29.95
http://pragmaticprogrammer.com/titles/dlret

Competitive Edge

Need to get software out the door? Then you want to see how to *Ship It!* with less fuss and more features. And every developer can benefit from the *Practices of an Agile Developer*.

Ship It!

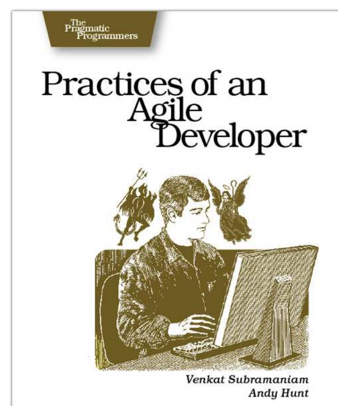
Page after page of solid advice, all tried and tested in the real world. This book offers a collection of tips that show you what tools a successful team has to use, and how to use them well. You'll get quick, easy-to-follow advice on modern techniques and when they should be applied. **You need this book if:** • You're frustrated at lack of progress on your project. • You want to make yourself and your team more valuable. • You've looked at methodologies such as Extreme Programming (XP) and felt they were too, well, extreme. • You've looked at the Rational Unified Process (RUP) or CMM/I methods and cringed at the learning curve and costs. • **You need to get software out the door without excuses**



Ship It! A Practical Guide to Successful Software Projects
Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. $29.95
http://pragmaticprogrammer.com/titles/prj

Practices of an Agile Developer

Agility is all about using feedback to respond to change. Learn how to apply the principles of agility throughout the software development process • Establish and maintain an agile working environment • Deliver what users really want • Use personal agile techniques for better coding and debugging • Use effective collaborative techniques for better teamwork • Move to an agile approach



Practices of an Agile Developer: Working in the Real World
Venkat Subramaniam and Andy Hunt
(189 pages) ISBN: 0-9745140-8-X. $29.95
http://pragmaticprogrammer.com/titles/pad

Cutting Edge

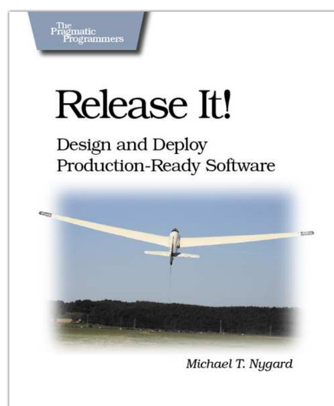
Now that you've finished your project, are you sure that it's ready for the real world? Are you truly ready to *Release It!* in this crazy world?

Interested in Ruby on Rails, but don't want to learn another framework from scratch? You don't have to! *Rails for Java Programmers*leverages you and your team's knowledge of Java to quickly learn the Rails environment.

Release It!

Whether it's in Java, .NET, or Ruby on Rails, getting your application ready to ship is only half the battle. Did you design your system to survive a sudden rush of visitors from Digg or Slashdot? Or an influx of real world customers from 100 different countries? Are you ready for a world filled with flakey networks, tangled databases, and impatient users?

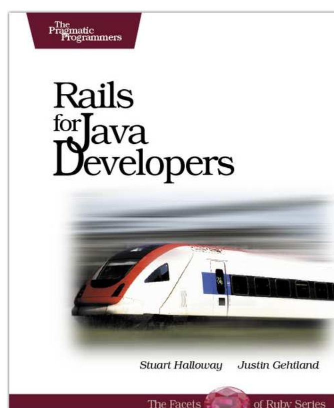
If you're a developer and don't want to be on call at 3AM for the rest of your life, this book will help.



Design and Deploy Production-Ready Software
Michael T. Nygard
(368 pages) ISBN: 0-9787392-1-3. $34.95
http://pragmaticprogrammer.com/titles/mnee

Rails for Java Developers

Enterprise Java developers already have most of the skills needed to create Rails applications. They just need a guide which shows how their Java knowledge maps to the Rails world. That's what this book does. It covers: • The Ruby language • Building MVC Applications • Unit and Functional Testing • Security • Project Automation • Configuration • Web Services This book is the fast track for Java programmers who are learning or evaluating Ruby on Rails.



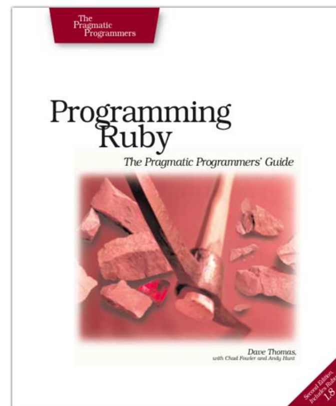
Rails for Java Developers
Stuart Halloway and Justin Gehtland
(300 pages) ISBN: 0-9776166-9-X. $34.95
http://pragmaticprogrammer.com/titles/fr_r4j

Facets of Ruby Series

If you're serious about Ruby, you need the definitive reference to the language. The Pickaxe: *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition*. This is *the* definitive guide for all Ruby programmers. And you'll need a good text editor, too. On the Mac, we recommend TextMate.

Programming Ruby (The Pickaxe)

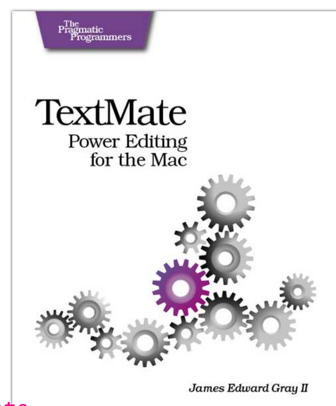
The Pickaxe book, named for the tool on the cover, is the definitive reference to this highly-regarded language. • Up-to-date and expanded for Ruby version 1.8 • Complete documentation of all the built-in classes, modules, and methods • Complete descriptions of all ninety-eight standard libraries • 200+ pages of new content in this edition • Learn more about Ruby's web tools, unit testing, and programming philosophy



Programming Ruby: The Pragmatic Programmer's Guide, 2nd Edition
Dave Thomas with Chad Fowler and Andy Hunt
(864 pages) ISBN: 0-9745140-5-5. $44.95
http://pragmaticprogrammer.com/titles/ruby

TextMate

If you're coding Ruby or Rails on a Mac, then you owe it to yourself to get the TextMate editor. And, once you're using TextMate, you owe it to yourself to pick up this book. It's packed with information which will help you automate all your editing tasks, saving you time to concentrate on the important stuff. Use snippets to insert boilerplate code and refactorings to move stuff around. Learn how to write your own extensions to customize it to the way you work.



TextMate: Power Editing for the Mac
James Edward Gray II
(200 pages) ISBN: 0-9787392-3-X. $29.95
http://pragmaticprogrammer.com/titles/textmate

Pragmatic Starter Kit

Version control. **Unit Testing**. **Project Automation**. Three great titles, one objective. To get you up to speed with the essentials for successful project development. Keep your source under control, your bugs in check, and your process repeatable with these three concise, readable books from The Pragmatic Bookshelf.

Visit Us Online

Unit Testing in C# Home Page
http://pragmaticprogrammer.com/titles/utc2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragmaticprogrammer.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragmaticprogrammer.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragmaticprogrammer.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: http://pragmaticprogrammer.com/titles/utc2.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com