The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

Issue #18
December 2010

Contents

FEATURES

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Up Front

The Ruby Issue

by Michael Swaine

This is our Ruby issue, celebrating ten years of RubyConf, the tenth anniversary of the Pickaxe Book [U1], and the latest version of the language.

The Ruby programming language, as everyone from Kansas to Oz knows, was created by Yukihiro "Matz" Matsumoto in 1993, was first seen in the wild in 1995, and finally came to the attention of forward-thinking American programmers around 1999. With the arrival of the Rails framework in 2005, Ruby got famous, but it hasn't forgotten its old friends, even as it captivates new friends with its magic.

We've put on our Ruby slippers for this issue, so get ready to experience the magic.

Specifically, Pragmatic Dave Thomas explores the magic to be found in Ruby 1.9.2. Named matches and Enumerators and multinationalization, oh my! Dave shows why you ought to be using Ruby 1.9.2 right now. And Paolo Perrotta gives three reasons why you should be using Bundler right now to manage your Ruby gems.

In addition to the articles by Dave and Paolo, Ian Dees starts a new series of features on everyday JRuby, and Chad Fowler talks with us about Ruby from its early days to its bright future, in a wide-ranging must-read interview.

If you're an experienced Rubyist, you'll find much to enchant you here. And if you've been waiting for an entrée to the magic land of Ruby, consider this issue your engraved invitation. Welcome. You're not in C++ anymore.

But Wait, There's More

In addition to our Ruby goodies, this issue has a few other gems. Tim Ottinger and Jeff Langr continue their series of agile articles with a thoughtful essay on the value of "Cohesive Software Design." Jonathan Rasmusson's "Way of the Agile Warrior" morphs for the month into the "Way of the Spartan Warrior." Andy Hunt explains why you should seek "The Perfect Deadline." And Dan Wohlbruck's series on technology history wraps up the year with a look back at the era of the personal computer.

Also, as usual, we troll the Twitterstream for Choice Bits and keep you up to date with our Events Calendar, while John Shade offers his view on dice and shell games and the twisted topology of technological progress.

External resources referenced in this article:

[U1] http://pragprog.com/refer/pragpub18/titles/ruby3/programming-ruby-1-9

Cohesive Software Design

Cohesion makes code easier to understand, debug, and test.

by Tim Ottinger, Jeff Langr

Cohesion—the principle that "things that belong together should be kept together"—makes code easier to understand, debug, and test.

The *Agile In A Flash* deck we're developing for the Pragmatic Bookshelf forced us to be very concise in expressing ideas about agile development, since we were constrained to very small spaces (5"x7" index cards). But we are passionate about some large ideas that we simply could not fit on the cards. We also had a limited number of cards, so we emphasized agile topics over broader topics such as design or the theory of constraints.

Our intent with this *PragPub* article series is to provide more depth than cards can ever do, and to help provide some of what we think are critical, missing pieces. We'll start with design. We believe that agile endeavor in the absence of good design sense is folly and a recipe for disaster.

The biggest ideas in software are:

1. Cohesion - proximity should follow dependency
2. Coupling - dependency should be minimized
3. Abstraction - hiding details simplifies code
4. Volatility - changes tend to cluster.

To some developers, these big ideas may sound like academic theories and CS dogma, disconnected from the real world of releasing new code every few days, every few weeks, or every few hours. But nothing could be further from the truth. Programmers famously complain and agonize over the pile of code they have to deal with, yet these four ideas succinctly describe the situation we are in, how it came to be, and how we can move forward. If we understand how these ideas apply to us, we can improve the code we write so that our software becomes increasingly workable.

We'll talk about cohesion this month, and follow up with the other three ideas in later articles.

What is Cohesion?

Cohesion tells us that *proximity follows dependency*—simply put, "things that belong together should be kept together." The classic OO design involves grouping data and the functions that act on it. We can understand cohesion as a measure of how exclusively an object's methods interact with its data and how exclusively an object's data is used by its methods.

Cohesion has been egregiously violated when an object's function does not use any of the object's data, when an object contains data that is not used by its own functions, or when an object has a function that deals only with the data of some other object. Less egregious violations are common, where a function deals with some of its object's data but deals also with the data or

functions exposed by another object. Code smells such as *feature envy* and *primitive obsession* are tip-offs to a lack of cohesion.

Lack of cohesion can also be apparent from an external perspective: If you find that approaching any system behavior via one path (command, screen, or web page) produces different results than approaching the same behavior from a different path, you've been bitten by duplication as an effect of poor cohesion.

Good cohesion makes it possible to find functions easily. If methods dealing with an address are located in the Address class, someone working with an address can easily find the method they need and call it, instead of re-implementing it from scratch. If functions that deal with an address are buried inside an mail-authoring component, a programmer working outside of that component is unlikely to find them. He may end up rewriting the needed behavior into the function he is working on so that afterward it is *still* not in the Address class. Later his coworkers will re-implement it again. Without cohesion, duplication propagates.

Cohesive code forms tighter, more testable objects. Testability has gained a lot of prominence due to test-driven development (TDD), acceptance TDD (aka ATDD), behavior-driven development (BDD), and their fellow travelers. A method with poor cohesion may do many things in the course of trying to accomplish some other goal entirely. These side-effects make it hard to test. The non-cohesive method may also bury some functionality in a maze of if/else statements that require careful and complex setup. A cohesive class is comparatively trivial to unit test.

Cohesion Simplifies Objects

In a system with primitive obsession, related data is passed around as loose variables to any number of functions, not combined into an abstract data type or class. These functions operate directly on the data—creating disjoint (non-cohesive) code—and will often re-implement common operations on the data in the course of fulfilling their own behaviors.

As cohesion is improved the loose variables are pulled together into a new class, and the functions that operated on the loose data are also moved into the class. This simplifies the old data-using methods, reduces code duplication, and establishes a home where others can expect to find similar functions. It greatly shortens parameter lists for many functions. A properly cohesive design can also allow for optimizations like lazy loading or caching that wouldn't be possible if the functionality were still spread across the application.

Closely related to cohesion, the Single Responsibility Principle (SRP) tells us that each class should do one thing and one thing only. Following SRP results in a cohesive class structure in which each class is a "conceptual center" in the application. You know why you should go there, and what it should do, and why it should change. You can understand it as a whole. It is deucedly hard to give a meaningful name to an amorphous blob of functionality, but when a class does one thing it is relatively easy to name it.

Another way of violating cohesion is seen in packaging structures that are defined according to some architectural ideal instead of being based on common

dependencies among objects. Poor packaging makes systems harder to refactor, harder to comprehend, and slower to build and deploy than systems that are packaged according to the principle of cohesion.

An Example

The following bit of code is from a free-source game. It exhibits a classic way to violate cohesion: throwing everything into an application startup class. In this case, we have a number of hints that there's a problem (other than the class source file being over 3000 lines long). We can take a look at the fields on the class:

```
private StringTokenizer StringT;
protected RiskController controller;
protected RiskGame game;
private PrintWriter outChat = null;
private BufferedReader inChat = null;
private ChatArea chatter = null;
private Socket chatSocket = null;
private ChatDisplayThread myReader = null;
private int port;
protected String myAddress;
private ByteArrayOutputStream Undo = new ByteArrayOutputStream();
protected boolean unlimitedLocalMode;
private boolean autoplaceall;
private boolean battle;
private boolean replay;
protected Vector inbox;
protected ResourceBundle resb;
protected Properties riskconfig;
```

The class has numerous reasons to change, an obvious violation of the SRP and thus an indicator of poor cohesion. We made some assumptions about what this class does, based solely on analyzing the field names and types:

- loads information from or stores information to a properties file
- contains flags for managing game properties (replay, battle, autoplaceall)
- manages a socket-based server

In other words, about the only thing all these variables have in common is that they appear in the same program.

A look at the methods confirms that these and many more responsibilities are munged together. The bulk of the class code appears in a single 1800-line-long method named GameParser, a name that turns out to not be so precise. This method serves not only the purpose of tokenizing input coming from the socket server (see the elided example below), but also is in charge of interacting with the game controller object that embodies most of the game rules (not shown).

```
String T = new StringTokenizer( message );
String Addr = GetNext();
if (Addr.equals("ERROR")) { // server has sent us a error
   // ...
}
else if (Addr.equals("DICE")) { // a server command
  // ...
}
else if (Addr.equals("PLAYER")) { // a server command
 // and on and on...
```

The game built on this source base is at least five years old, and continues to be maintained by its sole programmer with several releases a year. (We chose this example as a fairly typical source of cohesion violation, despite its being a sole-programmer effort. It exhibits characteristics common to most source bases we've ever worked on. Yet it's actually an enjoyable and now fairly solid game.) The programmer is no doubt intimately familiar with the code, and has the advantage of not working in a team of seven or twenty or two hundred programmers. Still, the release log is dominated by notes that start with the word "FIXED," particularly in the first few years of release.

Instead of continuing to conjecture about another programmer's experience, we'll relate our observations of the impact of poorly cohesive code in larger, more enterprise-y source bases. Over time, original programmers lose familiarity with the continually growing code base and sometimes leave the project. The shortcomings of the source base become more visible and apparent to everyone who remains.

We learn of more defects. We are frustrated by the difficulty of setting up just the right conditions to replicate the defects in unit tests so that we can fix them. We spend more time, far too much time, in simply understanding what's going on in a disjoint class. We spend still more time determining the total effect from even trivial modifications on line 200 of an 1800-line monolith. The ugliness causes no end of other nuisances; for example, having so many variables in the same address space makes automatic code completion less satisfying.

It doesn't have to be this way. As stated in Uncle Bob's book *Clean Code*, the first rule of functions, and also of classes, is that they should be small. The second rule is that they should be smaller than that. While it's possible to go overboard with a notion of ever-smaller, it's actually very rare, and it's pretty obvious when methods are no longer pulling their weight. Don't fear tiny methods and classes.

Try it. As a simple exercise, break one or more longer (20+ line) methods up by simply extracting small, logically related chunks of code. In Java or C#, take advantage of your IDE to make this a reasonably safe exercise. Sometimes you can use "guiding comments" as hints on what bits of code you might extract. Once you've extracted the methods, re-examine them. Do they really depend on the core concepts of the class in which they're defined? Or could you potentially move them to another existing class? Or do they represent a new abstraction entirely that you can embody in a separate class? (We'll provide some code examples in an upcoming article on abstraction.) Can you spot simple two- or even one-line repetitions in the longer methods that you could extract to a simpler abstraction, possibly generating some valuable reuse?

Cohesion is a practical issue, and not at all a matter of some abstract sense of design purity. When proximity follows dependency, your system becomes easier to use, easier to fix, and sometimes easier to optimize. Work mindfully, and see where your new appreciation of cohesion takes you this month.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U1] with Tim, he's written another couple books, *Agile Java* and *Essential Java Style*, contributed to Uncle Bob's *Clean Code*, and written 90+ articles on software development. Jeff runs Langr Software Solutions from Colorado Springs, where he also pair-programs full-time as an employee of GeoLearning.

About Tim

Tim Ottinger has over 30 years of software development experience coaching, training, leading, and sometimes even managing programmers. In addition to Agile in a Flash [U2], he is also a contributing author to *Clean Code*. He writes code. He likes it.

Send the authors your feedback [U3] or discuss the article in the magazine forum [U4].

External resources referenced in this article:

[U1] http://www.pragprog.com/refer/pragpub18/titles/olag/Agile-in-a-flash

[U2] http://www.pragprog.com/refer/pragpub18/titles/olag/Agile-in-a-flash

[U3] mailto:michael@pragprog.com?subject=Agile-cards

[U4] http://forums.pragprog.com/forums/134