The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

* Grokking Pattern Matching and List Comprehensions
* Everyday JRuby
* Code Coupling
* Rediscovering QA
* When Did That Happen?

Issue #19
January 2011

Contents

FEATURES

Grokking Pattern Matching and List Comprehensions 10
by Bruce Tate

Bruce explores two powerful features of modern programming languages that can make your code more beautiful and you more productive.

Everyday JRuby 16
by Ian Dees

Wherein Ian creates a simple game and then shows you several ways to deploy it.

Code Coupling 24
by Tim Ottinger, Jeff Langr

Those big software design concepts like coupling, cohesion, abstraction, and volatility have real practical value. In this article, Tim and Jeff talk about what coupling is, why it's necessary, and how you can reduce it to just that necessary amount.

Rediscovering QA 29
by Chris McMahon

Software Quality Assurance is more than testing. The breadth of knowledge necessary for really good QA work are surprisingly broad.

When Did That Happen? 32
by Dan Wohlbruck

Dan continues his series on the history of technology with a look at the index register.

Up Front

We Friend Your Curiosity

by Michael Swaine

What does it mean that this year began with Facebook surpassing Google in hits? So large a fact, distilled from so many individual human choices, must mean something about what it is to be human in the 21st century. But what?

Does it mean that we're more friendly than curious?

If so, that might be a good thing, but here at the start of 2011 and of this issue of *PragPub*, I'm banking on your curiosity.

If you have a healthy curiosity about programming languages, you'll enjoy Bruce Tate's exploration of language features that can make your individual coding style more powerful and artful.

If you're curious about how you can get a better view of the big picture in your development work, seeing it whole and from the user's perspective, you'll appreciate Chris McMahon's thoughts on software quality assurance, a discipline that doesn't always get enough respect or attention.

Jeff Langr and Tim Ottinger will satisfy your curiosity about one particular aspect of agile development, Ian Dees' series on everyday JRuby will enlighten you about methods for sharing your code, Dan Wohlbruck will satisfy your history itch with a shamelessly nerdy retrospective on the index register. Andy Hunt will tell you Why Johnny Can't Be Agile, and John Shade will share his take on the phenomenon of wikileaks.

So that's what's in this issue. In case you were curious.

Code Coupling

Reducing Dependency in Your Code

by Tim Ottinger, Jeff Langr

When it comes to couplings, there is strength in weakness.

Last month we listed the four biggest ideas in software: cohesion, coupling, abstraction, and volatility. Vowing to tackle one at a time, we focused on how lack of cohesion creates challenges in software. This month we continue through the idea list, moving on to the ever-present alliterative pair-partner of cohesion, *coupling*.

We learned that cohesion is to be maximized, so that *proximity follows dependency*. Where coupling is concerned, our rule is that dependency should be minimized. We'll tell you why and how.

Definition

When we talk about coupling in this article, we are referring to "attachments" required due to the dependencies between modules in an object-oriented (OO) system. A module can be a class or an aggregation of classes (a package). The term *coupling* can refer to other dependencies, such as those between methods, but we're not interested in those here.

A dependency exists when code in one class refers to another class—via any of several possible mechanisms:

- a field
- an argument
- constructed within a function
- inheritance or mix-in
- shared knowledge

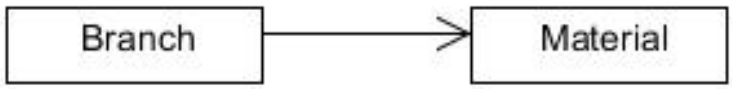
When you cannot compile or operate module A without the immediate presence of module B, module A is dependent upon module B. Dependency is coupling. More importantly, if a change in module B could cause breakage in module A, A is coupled to B.

Coupling among software components is reasonable and necessary. If a part (class, module, library) of a system we're writing has no connection to any other part, we typically call it "dead code" and delete it. All useful code has some coupling.

Simple Dependency

We need a Branch class to represent the various physical buildings in which library patrons can find books or other materials. We need a Material class to represent an item that patrons wish to borrow. A Branch object must contain a collection of Material objects. (Let's not confuse this with database design,

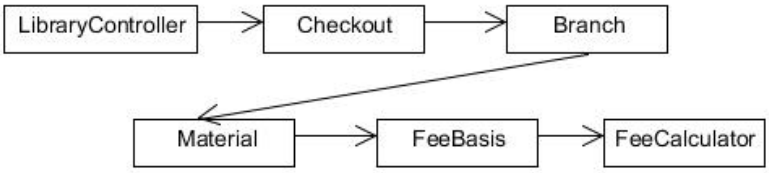
in which case a Material might have a back reference to a Branch table.) Figure 1 shows how we diagram such dependencies using UML.



Without Material, Branch has no reason to live; Branch is dependent upon Material. Any changes to the definition of Material might impact Branch. If you change the Material class, you'll need to re-test both the Material and Branch classes before releasing the change. On the other hand, changes to Branch are not important or interesting to Material. We can say that effects flow against the direction of the dependency arrows. *The problem with couplings is not their existence or even necessarily their number, but their potential to cause dependent code to break.*

Transitive Dependency

More trouble comes when there are many layers of dependency (see Figure 2).



Dependence is transitive. To understand the *full* impact of a dependency, follow the chain backward. For example, a small code change to the details of how fees are calculated in FeeCalculator has the potential to impact LibraryController! You've no doubt encountered this effect: you make a change in one corner of the system and it breaks code in a far, far distant corner. You might not even spot such a defect, particularly if you don't do extensive regression testing with each release. That of course means your unfortunate customer will likely be the one who spots the defect.

Unit testing, one of our favorite things to do (and do *first* of course, i.e. using TDD), also becomes far more difficult with deep dependency chains.

If you want to test the FeeCalculator class, you simply create a new instance:

```
var sut = new FeeCalculator(baseRate);
```

Since it has no dependencies, you're ready to call methods on it for verification purposes.

Tests for Material, on the other hand, will at some point require you to also construct an instance of FeeBasis, which requires an instance of a FeeCalculator. Tests of Branch will require Materials, and so on. By the time you arrive at the most dependent class, that insecure fellow who cannot live without a host of other objects, you may find yourself writing dozens of lines of code to create and populate associated objects.

Structural Dependency

Staying with the same example, if classes that use the LibraryController need to access the FeeCalculator by traversing Checkout, Branch, Material, and FeeBasis, then we have a structural dependency. If we ever collapse or extend the structure of that part of the application, code in all the places that are aware of the structure of the application will likely fail in one way or another. Hopefully it will be at compile time.

Implicit Dependencies

A diligent programmer can easily trace explicit dependencies between code modules as in the chain from LibraryController to FeeCalculator above. *Implicit* dependencies are rather trickier. When different parts of the program share knowledge, in violation of what we discussed last month about coherence, they will exhibit an implicit dependency.

Say that your auditor wants you to report the specific fee calculation method by name. Sadly, when the fee is calculated, the algorithm name is not recorded. You could change the fee transaction to include the algorithm name, but that means a change to a core business object and also to the database. Luckily, you remember that only the large payment calculator can levy fees as large as $23.00. You can infer the calculation method!

```
if Fee.amount > 23:
  calc_method_name = "large payment"
```

It works! You can now produce your report without touching/damaging existing code in the rest of the system! The problem is that now you have an implicit dependency on the limits of the large payment calculation. Maybe it works for now, maybe it will work for months or years, but it is not guaranteed always to work. If the calculation ever changes, this code will be quite broken, even though the fee report does not explicitly depend on the large payment fee calculator.

When details escape the class where they "belong" (see our previous article, "Cohesive Software Design"), we refer to them as leaky abstractions.

Fee.amount is a primitive integer in the example above. The authors implicitly assumed a currency. If they are our countrymen, it's a sure bet they're thinking US Dollars. And that might be all right, if there is a business rule in the system that all amounts are given as integer dollars, but it is a shared assumption.

Couplings are particularly troubling when they mix concerns that should be independent. If a calculation in the bowels of the system is written to pop up a modal dialog box, it couples calculation to user interaction. So much for calculating values in batch mode!

All implicit couplings seem convenient when first introduced, but later become reasons that the code cannot be easily diagnosed, repaired, or extended. In a small web app we've been working on, we spotted a shared bit of knowledge about file locations and URL construction. This information appeared in utilities, UI, and in the core class model of the app. Once we realized that we had a shared secret across modules, we realized we'd lost cohesion and had created implicit couplings that were going to bite us. Some redesign was in order.

How often do we hear developers saying that they would love to make an improvement to the code structure, but that it would take too long, cost too much, and risk too much? Typically, the programmer sent to resolve the problem must write the code that the original author avoided. Unfairly, the original author may be initially credited with quick turnaround, when in reality he did an incomplete job that his colleagues must finish for him.

Solutions

We need a way to maintain necessary couplings, but reduce the strength of the couplings so that a change in a depended-upon class does not cause rippling changes or failures throughout the system.

Prefer explicit couplings to implicit couplings. Increase cohesion so that one class becomes the single point of truth for a given fact. By doing so, we reduce the ability of implicit couplings to distribute errors throughout the code base. This is perhaps easier to say than do, but it goes a long way toward preventing "step back" errors.

If we can reduce our use of a class to a smaller set of method signatures, then we are depending on an interface rather than on the full implementation of the class. This is a weaker dependency that may replace couplings to a number of concrete (possibly derived) classes. This is the concept behind many "inversion of control" frameworks. The dependency on an interface also makes it a snap to replace a production concrete implementation with a stub, greatly simplifying the effort required to code unit tests.

It's better yet to depend on fundamental, unchanging interfaces on objects. For instance, if we can rework the code to treat the `FeeCalculator` as a black box into which we pass a rental instance and receive a fee object, we can use the facts the calculator provides to us without depending on how it does its work. This weaker dependency provides us with a kind of "dependency firewall." Abstraction (to be covered later) allows us to tolerate change.

Structural couplings are difficult to deal with, because navigation is frequently necessary and it has to go somewhere. The most common ways to deal with this are the Law of Demeter [U1] or the use of special classes (with names including words like Gateway or Repository) that will do navigation for us with methods like `Repository.FeeCalculatorFor(Material)`. We trade a dependency on the broader structure of the system for dependency on a single class that hides those dependencies for us.

It is common to introduce additional mechanisms (interfaces, abstract base classes, facades, repositories, navigation functions, etc.) to help us manage troublesome couplings. It adds some complexity to our applications to have these extra parts, but the additional mechanisms are trade-offs we willingly make to keep the system from degrading due to unnecessary strong couplings.

Summary

Coupling is necessary, it makes our code useful, but it can also make it fragile. By seeking weaker couplings, we can reduce code breakage in our systems. As a result, we'll spend less time tracking down weird problems and more time writing and polishing new features.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U2] 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 [U3], he is also a contributing author to *Clean Code*. He writes code. He likes it.

Send the authors your feedback [U4] or discuss the article in the magazine forum [U5].

External resources referenced in this article:

[U1] http://pragprog.com/refer/pragpub19/articles/tell-dont-ask

[U2] http://www.pragprog.com/refer/pragpub19/titles/olag/Agile-in-a-flash

[U3] http://www.pragprog.com/refer/pragpub19/titles/olag/Agile-in-a-flash

[U4] mailto:michael@pragprog.com?subject=Agile-cards

[U5] http://forums.pragprog.com/forums/134