The Craftsman: 46
Brown Bag III
Squaring the Circle

Robert C. Martin
5 January 2006

...Continued from last month.

Aug, 1944.

"How do you push thirteen quadrillion tons of loosely packed gravel and snow to one side?" Linus Pauling asked.

"I don't think you do." Replied Jan Oort. "It would be like trying to push six thousand cubic kilometers of feathers. There's nothing to push against. The pusher just penetrates the pile without exerting any useful force against it."

"That's exactly the problem." Pauling agreed. "If we had 50 years we could pull it aside with the gravity of a massive ship. But Clyde will be here in fifteen years. We can't push it aside, we can't dodge it, and we can't destroy it."

Wednesday, 27 Feb 2002, 1100

I was already waiting in the conference room when Avery walked in. He had Jerry with him. We greeted each other, and then got down to business.

"I hear you guys have been investigating dependency cycles." Jerry blurted.

"Right." I acknowledged. "Avery showed me the VISITOR pattern yesterday, and that led to a discussion of dependency cycles. He described to me how such cycles make the build order ambiguous, but he also mentioned that there were other costs."

Jerry nodded. "Sure, there are all kinds of problems with dependency cycles. For example, if you have two components in two different jar files, and there is a dependency cycle between those components, then you cannot deploy those two jar files independently."

Both Avery and I nodded. That made perfect sense.

"Also, any class that depends on a member of a cycle, depends upon *every* member of the cycle."

I thought about this for a second, and then said: "You mean transitively?"

"Of course." Jerry replied. "But transitive dependencies are dependencies. If A depends on B, and B depends on C, then A depends on C."

"OK, I see that." I said. "And I understand that all them members of a cycle must be compiled and deployed together. But is there any other harm?"

"The number of dependencies in a cycle" Jerry continued "is related to the square of the number of classes in the cycle."

"Uh..." I said intelligently.

Jerry took a professorial stance. "Consider a cycle of A, B, and C. A depends on B which depends on C which depends back on A. How many dependencies are there?"

"I counted three in your description." I said.

"Yes, but there are actually six. AB, BC, CA, and the transitive dependencies: BA, CB, AC."

"OK, I guess." I frowned.

"Perhaps you don't think A really depends on C?" Jerry challenged.

"Well, it's indirect at best." I hedged.

"Consider that `C` has a method `f(int a)`, and you change it's signature to `f(double a)`. Is it possible, even likely, that you'll have to change the source code of B?"

"Sure." I said. "`B` probably calls `C.f`, and so B is likely to need some changing. Probably some `int` inside of `B` will have to change to a `double`."

"Right. Now what about `A`?"

"What about it?"

"Is there some `int` inside of `A` that might need to change to a `double`?"

"Er... Maybe."

"In fact, this kind of backwards chaining between modules is very common! Changes made at leaves of a dependency tree often propagate back along the branches."

"OK." I sighed. "For the sake of this argument, let's say that changes to `C` will likely cause changes to `A`."

Jerry smiled, while Avery remained oddly silent. "OK, now what if there are four modules, A, B , C, and D. How many dependencies are there?"

I drew a square and connected the vertices, and counted. "Twelve, I said."

"And for five modules?"

I imagined the pentagon in my head and said: "Twenty."

Avery jumped in: "The general formula is $\mathbf{n^2-n}$."

Jerry nodded. "Both right. So the number of dependencies in a cycle is $O(n^2)$."

"OK, I see that, but why is that significant? Why do I care that the number of dependencies in a cycle is related to the square of the number of modules in that cycle?"

Jerry smiled. "What is a software system?"

I rolled my eyes. This topic was leaving the ship and going to some other galaxy. "A software system is a collection of classes."

Jerry smirked and said: "That will do for the moment. Now how do you understand that system?"

A little itch started at the back of my brain. "You understand that system by learning what the classes do, and how they interrelate – Oh!"

Jerry's smile got as wide and condescending as Jasper's. "Yeah! Cycles are *hard* to understand! And the bigger the cycle, the harder they are. And to make matters worse, every class that depends on a cycle, depends upon every class in that cycle; so they are hard to understand too."

"Hmmm. I said. I understand the math, but I'm not sure I agree with the premise. Is the difficulty in understanding a software system really based on the number of dependencies?"

"It's certainly a component." said Jerry. "I have seen systems that started out simple, but that became virtually un-maintainable in a very short time. Every time anyone tried to make a change, they broke the system in unexpected ways and in unexpected places. And for an unexpected module to break, it had to be dependent on the module that changed. I have seen development efforts start fast and furious only to grind to a near halt in a matter of months because the developers lost control of the dependencies. So yes, Alphonse, dependencies really are a significant factor in the understanding and maintainability of a system."

We all sat there looking at each other. Jerry was smiling, Avery was withdrawn, and I was pensive. Finally, I said: "If this is true, then it is a guilt-edged priority to keep dependency cycles out of our systems.

Jerry's smile slowly faded. "In Mr. C's organization, it *definitely* is. Especially cycles between packages!"

"How do you prevent them?" I asked.

"There are tools that find them for you. JDepend[1] will hunt through your whole Java project and find dependency cycles. And there is a plugin[2] for FitNesse that uses JDepend so that you can make acceptance tests that check for cycles."

I thought about that for a minute. "Cool! That means the build will fail if you have a dependency cycle. If you do that, you'll simply never get cycles because you'll have to fix them as soon as they happen!"

"Yeah. We haven't set that up yet on Dosage Tracking, but we will."

"OK, OK." Said Avery, suddenly getting into the conversation, "But what about the VISITOR? the VISITOR creates a cycle of dependencies. Is that bad?"

Jerry visibly gathered his thoughts into this new direction and then said: "Well, usually the cycles from the VISITOR pattern are so small that they don't cause much harm."

"But what if you wanted to break them anyway?" Avery said.

"Well, you could use the ACYCLIC VISITOR."

Avery and I both said: "The what?"

Jerry laughed and said: "Here, let me show you." And he pulled up the example of the `SuitInspectionReport` that we had used yesteray. He made a few changes to it, and then said: "There. That's the acyclic version of the VISITOR."

The first thing to notice is that the `SuitVisitor` has become a *degenerate interface* having no methods or variables. Interfaces like this are sometimes called *marker interfaces*.

```
public interface SuitVisitor {}
```

Next, notice that there are two new interfaces, one for the `MensSuit`, and one for the `WomensSuit`.

```
public interface MensSuitVisitor {
  void visit(MensSuit ms);
}

public interface WomensSuitVisitor {
  void visit(WomensSuit ws);
}
```

The `SuitInspectionReportVisitor` is the same as before, except that it now implements all three visitor interfaces.

```
public class SuitInspectionReportVisitor implements SuitVisitor,
                                                    MensSuitVisitor,
                                                    WomensSuitVisitor {
  public String line;

  public void visit(MensSuit ms) {
    line = String.format("%d MS A:%d, B:%d, C:%d",
                         ms.barcode, ms.ipa, ms.ipb, ms.ipc);
  }

  public void visit(WomensSuit ws) {
    line = String.format("%d WS A:%d, B:%d",
                         ws.barcode, ws.ipa, ws.ipb);
```

[1] www.clarkware.com

[2] http://butunclebob.com/ArticleS.UncleBob.StableDependenciesFixture

```
  }
}
```

Finally, the `MensSuit` and `WomensSuit` classes implement the `accept` method a little differently.

```
public class MensSuit extends Suit {
  int ipa;
  int ipb;
  int ipc;

  public MensSuit(int barCode) {
    super(barCode);
  }

  public void accept(SuitVisitor v) {
    if (v instanceof MensSuitVisitor)
      ((MensSuitVisitor)v).visit(this);
  }
}

public class WomensSuit extends Suit {
  int ipa;
  int ipb;

  public WomensSuit(int barCode) {
    super(barCode);
  }

  public void accept(SuitVisitor v) {
    if (v instanceof WomensSuitVisitor)
      ((WomensSuitVisitor)v).visit(this);
  }
}
```

The `SuitInspectionReport` uses the `SuitInspectionReportVisitor` in exactly the same way as before.

```
public class SuitInspectionReport {
  public String generate(SuitInventory inventory) {
    StringBuffer report = new StringBuffer();
    SuitInspectionReportVisitor v = new SuitInspectionReportVisitor();
    for (Suit s : inventory.getSuits()) {
      s.accept(v);
      report.append(v.line).append("\n");
    }
    return report.toString();
  }
}
```

I looked at this code for awhile and then said: "This breaks the cycle all right; and it retains all the benefits of the original VISITOR. So why wouldn't we always use this?"

Jerry scratched his head and said: "Well in a case like this the cycle in the regular VISITOR just isn't very harmful. Moreover, the VISITOR is a bit faster and simpler than the ACYCLIC VISITOR. So I'd probably opt for the simpler case. However, if there were a lot of classes and dependencies involved; I'd be tempted to break the cycle."

Our time was up, so we walked out of the conference room and back to our workstations. Jerry strode quickly ahead of us. When he was out of earshot I heard Avery murmur: "He ruined everything!"

To be continued...