The Craftsman: 28
Dosage Tracking V
An Encapsulation Break

Robert C. Martin
29 June 2004

...Continued from last month.

---

21 Feb 2002, 1100

As tensions in Europe rose and eventually blossomed into war, Clyde was all but forgotten by the public -- but not by Tombaugh. He continued to track Clyde, computing and re-computing its orbit. His results were published in the appropriate journals, and were noted with growing concern by scientists on both sides of the growing political divide. The odds for a collision with Earth, though still very low, continued to rise with every new measurement.

By early 1939, when it was clear to most scientists that they would soon be unable to communicate with their colleagues on the other side of the Atlantic, Tombaugh decided to hold an astronomical conference in Stockholm. The conference was very well attended, and not just by astronomers. It seemed that scientists of many disciplines felt that this conference could be their last chance for an international gathering.

Clyde was not a major topic of the formal conference, but around the bar, and in the lounges, the talk often drifted to: "what if?" These discussions weren't serious so much as they were entertaining; but they served to generate some interesting ideas. Some were about destroying Clyde, others were about deflecting it, and still others were about escaping to Venus, or Mars.

Despite the range of ideas, everyone agreed that the fundamental problem was energy. All these solutions required energies that were beyond our abilities to generate; and so they were all considered to be bar-room fantasies.

Then, on the last night of the conference, Leo Szilard, Neils Bohr, and Lise Meitner broke the news that their investigations into nuclear reactions indicated that such energies just might be within reach. In view of the political climate, it was clear to everyone there that this news was not all that good. A few stayed very late that night to discuss…options. They whimsically named themselves: The Stockholm Contingent.

Jerry, Avery, and I went to the observation lounge at the high-gee end of the gamma arm. We each got a cup of coffee and sat at a table watching the starbow whirl under our feet.

Jerry looked over the table at Avery and said: "So, Avery, you complained about my use of public variables in the test fixture we were writing."

Avery grimaced and said: "Well, it's not a very OO thing to do."

"Can you define OO for me?" Asked Jerry? "Why are public variables not OO?"

"Because OO is about encapsulation, and public variables aren't encapsulated."

"Do you think that OO requires all variables to be encapsulated?"

"Of course!" replied Avery.

Jasmine was at a nearby table with Adelade. She seemed interested in this conversation.

Jerry asked: "What bad thing happens if you have an un-encapsulated variable?"

Avery made a disdainful face and said: "Other programs could change your variables!"

"What if that is what you intend? What if you *want* other programs to change certain variables?"

"Well then there's something wrong with your design!" Avery proclaimed with righteous indignation.

Jasmine loomed over our table and said: "Oh pooh, Avery, that's just silly."

Avery hadn't noticed Jasmine until that moment. When he saw her, the expression on his faced changed to a mix between embarrassment and panic. He sputtered as though to answer her, but Jasmine kept right on talking.

"Look you meatballs, it's not the rules, it's the reasons. The aim of OO is to help you manage dependencies. Most of the time making your variables private helps you to do that. But sometimes private variables just get in the way."

Jerry butted in: "Now wait a minute, Jasmine. I agree that OO languages help you to manage dependencies, but the more important benefit is expressivity. It is easier to express concepts using an OO language."

"Yeah, yeah, sure, sure." Replied Jasmine. But it's not lack of expressivity that turns software systems into a tangled mass of interconnected modules; it's mismanaged dependencies that do that. I can create very expressive systems that are tangled all to hell. I agree that expressivity is important, but keeping the coupling low is much more important!"

"How can you say that? You're approach is so sterile! Where is the art? Where is the finesse? All you care about is that the dependencies go in the right direction."

"What is artful about nicely named modules that are coupled into a tangled mess? The art is in the structure, Jerry. The art is in the careful management of dependencies between modules."

"Structure is important, sure, but…"

It seemed to me that this was likely to go on for awhile, and I really wanted to know about the public variables, because they bothered me too. So I interrupted them. "Excuse me, Jasmine, Jerry, but what does this have to do with public variables?"

The two of them looked at Avery and me as though they just remembered we were there. Jerry said: "Oh, uh, sorry about that. This is an old argument between Jasmine and I. Look, we make variables private to tell other programmers to ignore them. It's a way of expressing to others that the variables aren't their problem, and that we want them to mind their own business."

Jasmine rolled her eyes. "Oh, here we go again! Look boys, (I didn't like being called a boy – especially by Jasmine. I saw Avery flinch too.) we make variables private to reduce coupling. Private variables are a firewall that dependencies cannot cross. No other module can couple to a private variable."

"That's such an ice-cold view" said Jerry. "Don't you see how dehumanizing it is?"

I wanted to stop this before it got going again, so I blurted out my next question: "OK, but when is it OK for a variable to be public?"

Avery seemed to recover himself at that. He looked at me and proclaimed: "Never!"

That stopped Jasmine and Jerry in mid-stride. They both turned to Avery and said: "Nonsense – Ridiculous".

"Look, Avery" said Jasmine "sometimes you *want* to couple to a variable."

"Right" Said Jerry. "Sometimes a public variable is the best way to communicate your intent.

"When is that?"

"Well, like in a test fixture." Jerry replied. "The data used by the fixture comes from one source, but the test is triggered by another. One party has to load the data, and the other party has to operate on it. We want to keep them separate."

"Right" said Jasmine. "It's all about the coupling. The important thing is that the source of the data is not coupled to the application being tested. The fact that the variables in the fixture are public is harmless."

"But what if someone uses them?" complained Avery?

"Who would?" Replied Jerry. "They are variables in a test fixture. Everybody on this project knows that those variables are under the control of FitNesse[1] and aren't for anyone else to use."

"But they might!"

Jasmine rolled her eyes and said: "Yeah, and they might take one of your private variables and make it public. What's the difference?"

Avery spluttered some more, but couldn't seem to face her down. So I asked the obvious question: "But couldn't you use setters and getters? Why make the variables public?"

"Exactly!" Avery splurted.

"Why bother?" Asked Jerry.

"Right," Said Jasmine. "It's just extra code that doesn't buy you anything."

"But I thought getters and setters were the right way to provide access to variables?" I replied.

"Getters and setters can be useful for variables that you specifically want to encapsulate." Said Jerry, but you don't have to use them for all variables. Indeed, using them for all variables is a nasty code smell."

"Right!" responded Jasmine with a grin. "PeeYew! There's nothing worse than a class that has a getter and setter for each variable, and no other methods!"

"But isn't that how you are supposed to implement a data structure?" I asked.

"No, not at all." Replied Jasmine. "A class can play two roles in Java. The first is as an object, in which case we usually make the variables private and provide a few getters and setters for the most important variables. The other role a class can play is as a data structure, in which case we make the variables public and provide no methods. Test fixtures are primarily data structures with a few methods for moving the data back and forth between FitNesse and the application under test."

"Right." Said Jerry. "A class that has getters and setters but no other methods isn't an object at all. It's just a data structure that someone has wasted a lot of time and effort trying to encapsulate. That encapsulation buys nothing in the end. Data structures, by definition, aren't encapsulated."

"Agreed." Said Jasmine. "Objects are encapsulated because they contain methods that change their variables. They don't have a lot of getters and setters because the data in an object remains hidden for the most part."

"Right." Said Jerry.

"Right." Said Jasmine.

And then the two of them smiled at each other in a way that made my stomach turn. Avery looked like he had just bitten into a lemon.

To be continued...

---

[1] http://fitnesse.org