Test-Driven Development

A Conversation with Martin Fowler, Part V

by Bill Venners

December 2, 2002

Summary

Martin Fowler, chief scientist at Thoughtworks, Inc. and author of numerous books on software design and process, talks with Bill Venners about the unhurried quality of test-first design and monological thinking, and the difference between unit and functional testing.

Over the last decade, Martin Fowler pioneered many software development techniques in the development of business information systems. He's well known for his work on object-oriented analysis and design, software patterns, Unified Modeling Language, agile software processes (particularly extreme programming), and refactoring. He is the author of *Analysis Patterns* (Oct. 1996), *Refactoring* (June 1999; coauthored with Kent Beck, et al.), *UML Distilled* (Aug. 1999; with Kendall Scott), *Planning Extreme Programming* (Oct. 2000; with Kent Beck), and the soon to be released *Patterns of Enterprise Application Architecture* (Nov. 2002), all published by Addison Wesley.

In this six-part interview, which is being published in weekly installments, Fowler gives his views on many topics, including refactoring, design, testing, and extreme programming. In Part I, Fowler makes the business case for refactoring and testing, and describes the interplay between refactoring, design, and reliability. In Part II, Fowler discusses design principles of avoiding duplication, separating presentation and domain logic, being explicit, and describes how refactoring depends on code ownership. In Part III, Fowler differentiates between planned and evolutionary design, suggests that focusing on superficial problems can lead to the discovery of substantial problems, and claims that doing a good job won't slow you down. In Part IV, Fowler discusses design decay, flexibility and reusability versus complexity, four criteria for a simple system, and interface design. In this fifth installment, Fowler describes the unhurried quality of test-first design, defines monological thinking, and distinguishes between unit and functional testing.

Designing in Small Steps

Bill Venners: When doing evolutionary design, are you designing interfaces in small steps, one piece of the interface at a time?

Martin Fowler: Yes. If I'm creating a `Money` class, I'll make `plus` work first before I even think about the interface of `multiply`. Just get `plus` to work. Don't think about anything else, just focus on `plus`. Come up with its interface. Come up with its implementation. Now we've done `plus`. Now let's do the next bit.

Bill Venners: I'm more comfortable with, OK I've got a `Money` class, what are its responsibilities? What services does it need to provide? Define the interfaces without implementing the code, and get that nice and easy to understand. And then you can write code, test, implement, test, implement. But I still tend to think more of partitioning the whole system into subsystems, and sub-subsystems, and designing interfaces in the chunks they'll be used, like classes.

Martin Fowler: The thing I like about taking small steps and writing tests first is that it gives me a simple to do list of the things I've got to do. At each end point I have a certain amount of closure. I say, OK, this stuff works. Check it in. It's all there. It does what it does and it does it correctly.

A Sense of Unhurriedness

Martin Fowler: There's an impossible-to-express quality about test-first design that gives you a sense of unhurriedness. You are actually moving very quickly, but there's an unhurriedness because you are creating little micro-goals for yourself and satisfying them. At each point you know you are doing one micro-goal piece of work, and it's done when the test passes. That is a very calming thing. It reduces the scope of what you have to think about. You don't have to think about everything you have to do in the class. You just have to think about one little piece of responsibility. You make that work and then you refactor it so everything is very nicely designed. Then you add in the next piece of responsibility. I previously used the kind of approach you describe. I'd ask, "What's the interface of this?" I've now switched and I much more prefer incremental design.

Bill Venners: Well, I'll try it.

Martin Fowler: In many ways, you have to just try the process out for a little while. It's best to try it out with someone who's done it before. Just do it.

Here's an example of incremental design that I ran into when writing *Patterns of Enterprise Applications Architecture Design*. I needed to create an example in a pattern about associative table mappings. If you have a many-to-many relationship in memory and you need to persist that to a relational database, it requires an extra link table, so you get three tables. There are multiple ways to pull that data back into memory from the database. There is a simplistic way of pulling the data back that requires a lot of select statements to be executed. There is also a faster way to pull it back. You can just pull the data back in a single select statement. But then it's a bit more awkward to get the data out and to unpack it into the various objects.

I created the example for that pattern using incremental design. I started by writing an example that was hard-coded for three particular tables and two particular classes. I didn't think about generalizing it. I just made a very specific, concrete example work. After I got that working and had it passing its tests, I started refactoring the example to make it more generally applicable. After a bit of time refactoring, I had a general mechanism. All I had to do was write a tiny mapping class, and I could make the example work with any tables and classes. I had a general purpose structure.

I found it much easier to do the concrete example first, then refactor that to the abstract example, than to come up with the abstract example first and then apply that to the concrete case. I also found it much more calm and unhurried, even though the process went pretty swiftly. I constantly knew where I was and where I was going. I felt much more in control of the situation. I didn't have that, "Am I ever going to make this code work?" feeling.

Monological Thinking

Martin Fowler: A phrase I've coined for Kent's new book is *monological thinking*. By monological, I mean using one style of thought, one style of logic, at any one time. When I was creating the specific example, I was just thinking about how to make a very concrete example work. When I moved into a refactoring mode, I was thinking about how to abstract the working concrete example. I wasn't thinking about how to make the example work and how to make an abstract solution at the same time. I made it work, then I abstracted it. I felt it went quicker and was a much more calm and pleasurable experience.

Bill Venners: It sounds like incremental design is a way of dealing with complexity given our limited brains. To what extent do you think whether you write tests first or second is a personality choice? Are some people by their nature more suited to this style of dealing with complexity versus another?

Martin Fowler: That may well be true. It could well be that there are personality differences. However, I think it is hard to tell at the moment because not enough people have tried the test-driven development approach. Test-driven development is common in the extreme programming community, but that's still a very small slice of the programming population.

I recommend you sit down with somebody whose done test-driven development, so that you do it pairing with somebody who knows that style. I think you'd get a much better appreciation of how it works that way, because it is so very counter-intuitive. Unfortunately we won't have time to do something like that, but I'd love to do that with you. I can almost guarantee that you'll say, "What we're going to take that small of a step? It's not worth going such a small step." And I'll say, "Just trust me. Do these tiny steps." I've seen it so many times. I remember watching someone pair program with Kent for

the first time. This guy had read up on XP and was pretty much in favor of it. He was very positive about XP already. There were times when his jaw was dropping at the tiny little moves Kent was making. He came out of it realizing that there is a whole style to test-driven development that he didn't expect.

Unit Testing

Bill Venners: What is the unit in unit test? Define unit test.

Martin Fowler: That's very difficult. To a first approximation, it's a class. But as you work with unit tests more, you begin to realize you're testing little areas of responsibility, and that could be a part of a class or it could be several classes together. I don't get too hung up about it, but I'd say if you're just starting out, think of unit tests as just writing a test case per class.

Bill Venners: And the reason you call it "unit" testing is that you're testing individual pieces that make up the program. To get a robust whole, you build it on robust parts, or robust units. To get the parts robust, you test them individually with unit tests.

Martin Fowler: That's the notion of where unit testing originally came from. Unit testing in XP is often unlike classical unit testing, because in XP you're usually not testing each unit in isolation. You're testing each class and its immediate connections to its neighbors.

Bill Venners: What's the difference between unit and functional testing?

Martin Fowler: The current name in XP circles for functional tests is acceptance tests. Acceptance tests view the system more as a black box and test more end to end across the whole system. You would have unit tests for individual classes in the domain and database mapping layers. In most of these unit tests, you might not even connect the database. You would stub the database out. But with the functional tests, which go end to end, you would want everything connected.

Bill Venners: When do you stop writing tests? You say in *Refactoring*, "There's a point of diminishing returns with testing, and there's a danger that by writing too many tests you become discouraged and end up not writing any. You should concentrate on where the risk is." How do you know where the risk is?

Martin Fowler: Ask yourself which bits of the program would you be scared to change? One test I've come up with since the *Refactoring* book is asking if there is any line of code that you could comment out and the tests wouldn't fail? If so, you are either missing a test or you've got an unnecessary line of code. Similarly, take any Boolean expression. Could you just reverse it? What test would fail? If there's not a test failing, then, you've obviously got some more tests to write or some code to remove.

Bill Venners: Sounds like you could actually automate that.

Martin Fowler: Actually, there is a program that does that. It's called JesTer. The problem is it takes a long time to run, because every time it makes one minor mutation it has to run the entire build test suite.

Talk Back!

Have an opinion about test-first development, incremental interface design, or unit testing? Discuss this article in the News & Ideas Forum topic, Test-Driven Development.

Resources

Refactoring: Improving the Design of Existing Code, by Martin Fowler with Kent Beck, John Brant, William Opdyke, and Don Roberts is at Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/0201485672/

To Be Explicit, an article by Martin Fowler first published in *IEEE Software*:
http://www.martinfowler.com/articles/explicit.pdf

Public versus Published Interfaces, an article by Martin Fowler first published in *IEEE Software*:

http://www.martinfowler.com/articles/published.pdf

JesTer, the tool mentioned by Martin Fowler that finds places in your code not covered by a unit test:
http://www.xpdeveloper.com/cgi-bin/wiki.cgi?JesTer

The Pragmatic Programmer: From Journeyman to Master, by Andrew Hunt and David Thomas, is at Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/020161622X/

IntelliJ IDEA, a Java IDE with refactoring support:
http://www.intellij.com/idea/

Eclipse, an open source IDE with refactoring support:
http://www.eclipse.org/

A catalog of summaries of refactorings mentioned in the book, *Refactoring*:
http://www.refactoring.com/catalog/index.html

A refactoring portal maintained by Martin Fowler contains links to refactoring tools and other refactoring sites:
http://www.refactoring.com/

Martin Fowler's links to extreme programming resources:
http://martinfowler.com/links.html

Articles written by Martin Fowler about XP and agile methods:
http://martinfowler.com/articles.html#agile

Patterns of Enterprise Application Architecture, by Martin Fowler is at Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/0321127420/

UML Distilled: A Brief Guide to the Standard Object Modeling Language, by Martin Fowler and Kendall Scott is at Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/020165783X/

Planning Extreme Programming, by Kent Beck and Martin Fowler is at Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/0201710919/

Analysis Patterns: Reusable Object Models , by Martin Fowler is at Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/0201895420/

Martin Fowler's website contains many articles, book chapters, and other information from Martin:
http://www.martinfowler.com/