

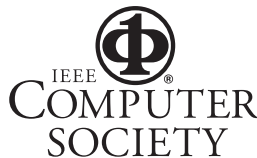


Driven to ... Discovering Your Design Values

Rebecca J. Wirfs-Brock

Vol. 24, No. 1
January/February 2007

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.



© 2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

For more information, please see www.ieee.org/portal/pages/about/documentation/copyright/polilink.html.

Driven to ... Discovering Your Design Values

Rebecca J. Wirfs-Brock

There will be variations of everything forever. ... Ideas don't disappear. They change form, they merge with other ideas.
—Bob Frankston, coinventor of VisiCalc

Today we have design or development approaches that are, for example, responsibility driven (RDD), test driven (TDD), behavior driven (BDD), domain driven (DDD), and model driven (MDD). Not all thought leaders in software development have been “driven”—Bertrand Meyer, for example, invented Design by Contract. But whether “driven” or not, these approaches all emphasize a core set of values and principles around which practices, techniques, and tools have emerged. A thoughtful designer should be able to pick and choose among practices without losing their essence. But not all practices are congruent. After stewing in this alphabet soup for years, I’m



been on exposing the common and complementary threads that are interwoven among various design practices.

Responsibility-driven design

So how can you integrate various practices without watering them down or muddling your thinking with too many considerations? It’s certainly easy if you have one belief system with one small set of coherent values and practices that guide your work. In 1989, Brian Wilkerson and I authored the paper “Object-Oriented Design: A Responsibility-Driven Approach.”¹ For better or worse, we started the trend of tagging design approaches as “driven.” To make our point, we oversimplistically divided object de-

sign approaches into two camps: those that focus first on structure and those that focus first on action (or responsibilities or behaviors). We argued that designers who first focus only on an object’s structure fail to maximize encapsulation. Thinking too early about structure makes it too easy for implementation details to bleed into class interfaces.

We contrasted two approaches for designing a `RasterImage` class that represented a rectangular grid of pixels. (We wrote this paper when raster technology was new, and at the time we both worked at Tektronix, a leading provider of graphics workstations.) With a data-first approach, we started defining our `RasterImage` class by declaring the image data structure and then adding methods to retrieve and set the image and query its dimensions. Voila!—a class where form and function were inextricably intertwined. The pixel grid data structure wasn’t considered a private implementation detail. Next, we demonstrated how a designer could think differently about the problem by asking, “What actions could this object be responsible for?” and “What information should it share with others?” This led us to first define operations for our `RasterImage` class to scale and rotate the image and access pixel values. The internal image representation, which we didn’t specify until after we’d defined the interface, was considered a private detail.

By consciously assigning most objects action-oriented responsibilities, you can design even seemingly data-centric objects to perform some actions as well as encapsulate structural details.

Hiding that structure makes those details easier to change. To us it seemed that the order in which a designer considers things profoundly affects the resulting design—even for a class as straightforward as `RasterImage`. To quote Samuel Alexander, the philosopher, “An object is not first imagined or thought about and then expected ... but in being actively expected it is imagined as future and in being willed it is thought.”

Since those early days I’ve added the notion of role stereotypes,² acknowledging that not all objects are active. Information holders—objects with responsibility for maintaining data—have a place in a design, too. But encapsulating their private details is important.

Test-driven design

RDD evolved in the highly interactive world of Smalltalk development, where developers routinely designed a little, coded a little, and tested a little in short cycles. The delightful tension of cycling between imagining what an object might do, building it, and then refining your ideas and cycling through your design again can lead to deep insights. This has led agile-programming thought leaders to promote test-driven development practices. Test-driven design emphasizes deciding on an interface and then writing code to test that interface, before implementing code to make the interface pass the test.

In *Test-Driven Development by Example*, Martin Fowler claims that TDD “gives you this sense of keeping just one ball in the air at once, so you can concentrate on that ball properly and do a really good job with it.”³ With a design-test-code-reflect-refactor rhythm, good code emerges alongside well-designed interfaces.

Linda Crispin explains that TDD isn’t really about testing.⁴ Instead, it’s a practice that gets you thinking about as many aspects of a feature as you can before you code it. With frameworks such as Fit and Fitness, TDD has extended beyond its initial focus on just developers to enable nontechnical people to write concrete examples of inputs and expected results in tabular form. Pro-

grammers write test fixtures that use these behavioral specifications to test the code.

As TDD practices have grown, new variants of them, along with newer testing frameworks, have emerged. Users of `jMock` use *mocks* that mimic unimplemented behaviors to drive out an appropriate distribution of responsibilities among collaborators.⁵ They don’t think that it’s just about testing, either. Mocking lets you incrementally design and build software, hypothesizing and refining your ideas as you go.

Behavior-driven design

BDD is another subtle refinement of TDD. BDD proponents firmly believe that how you talk about what you’re doing influences how you work. The focus is on writing small behavior specifications to drive out the appropriate design. As Dave Astels puts it, “A major difference is vocabulary. Instead of subclassing `TestCase` [as you would do using an `xUnit` framework], you subclass `Context`. Instead of writing methods that start with `test`, you start them with `should`.”⁶ Testing, to BDD proponents, connotes verifying code after it’s built. Instead, they want to encourage incremental design by writing small specifications, then implementing code that works according to spec.

Does every method warrant a contract? Probably not. Methods that don’t cause side effects probably don’t need contracts.

Design by Contract

In contrast, Design by Contract (DbC) has roots in formal specifications. To specify how they expect system elements to interact, designers write contracts specifying what must be true before a module can begin (preconditions), what must be preserved during its execution (invariants), and what it guarantees to be true after it completes (postconditions). You could specify contracts for components, services, or even individual methods. However, in practice, most contracts are written at the method level because existing programming languages and tools support work at that level. Writing contracts is easier if the languages and tools you use support them and you have good examples to emulate. The Eiffel language integrates contract support into the language and runtime environment; most other object-oriented languages don’t.

Before looking at `Contract4J`, a DbC tool for Java, I thought that specifying contracts for languages without built-in support would be clunky. However, using aspect technology, `Contract4J` automatically weaves aspect-specific contract tests, which are specified in method comments, into your running code. Contracts specified this way leave method code uncluttered with assertion statements and leave a nice documentation trail of how methods should be invoked.

If you choose to, you could apply TDD practices to understand your classes’ behaviors and then add contracts to methods whose behaviors you want verified at runtime. But I suspect these two communities differ considerably in their thinking. Some TDD proponents want to discourage after-the-fact verification, which to them seems antithetical to designing for quality. But adding contracts does tighten up how you use classes, theoretically making it easier to catch errors before they propagate.

When you change your design, sometimes contracts will naturally change, too. But once your design ideas settle down, you can finalize or add contractual details. But does every method warrant a contract? Probably not. Methods that don’t cause side ef-


fects probably don't need contracts. But I know I'd certainly find it easier to use class libraries if contract specifications were part of their documentation even if they weren't validated at runtime.

Domain-driven design

What about incorporating DDD ideas into your design practice? According to Eric Evans, DDD isn't a technology or methodology but "a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains" (www.domainlanguage.com/ddd/index.html). A central activity in DDD is searching for the language that experts use to talk about the problem and then literally reflecting that language in classes and services in a domain layer. Eric believes that, "If developers don't realize that changing code changes the model, then their refactoring will weaken the model rather than strengthen it." Creating a domain model is intricately tied to expressing it in working code. Domain-driven design is an active, ongoing process of expressing this domain language in code.

Model-driven design

In contrast, adherents of MDD (some call it *model-driven engineering* to avoid the Object Management Group trademarked term) first develop a platform-independent model of their system (usually in UML or a domain-specific language) before translation tools transform the model into platform-specific code. MDD practitioners strive to clearly represent system concepts and behaviors with the goal of producing an abstract model, not working code (the translation tools do that for them). This view of model building followed by transformation probably causes the great divide between MDD practitioners and other design schools—even though they share many common design values. After recently listening to and talking with several well-known MDD proponents who were discussing what constitutes well-designed classes, methods, and components, I found myself nodding in agreement with many of their design guidelines.

How you design should be based on your principles and values. Although a big division exists between those who believe the act of coding is what validates the design and those who don't, you can learn many things about good design from each. My mantra has always been, "Be open to new ideas and techniques that make me a better designer." I side with Canadian politician Dan Miller, who proclaims, "You know, we have our differences, everybody does, honest, real differences, but I do believe strongly that we as neighbors are drawn together far more than we're driven apart." 

References

1. R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," *Proc. 1989 ACM SIGPLAN Conf.*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 89), ACM Press, 1989, pp. 71–75.

2. R. Wirfs-Brock, "Characterizing Classes," *IEEE Software*, Mar./Apr. 2006, pp. 9–11.
3. M. Fowler, *Test-Driven Development by Example*, Addison-Wesley, 2003.
4. L. Crispin, "Driving Software Quality: How Test-Driven Development Impacts Software Quality," *IEEE Software*, Nov./Dec. 2006, pp. 70–71.
5. S. Freeman et al., "Mock Roles Not Objects," *Companion to 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 04), ACM Press, 2004, pp. 236–246; www.jmock.org/oopsla2004.pdf.
6. D. Astels, "A New Look at Test-Driven Development," http://blog.daveastels.com/files/BDD_Intro.pdf.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates and an adjunct professor at Oregon Health & Science University. Contact her at rebecca@wirfs-brock.com.

IEEE Pervasive Computing



delivers the latest peer-reviewed developments in pervasive, mobile, and ubiquitous computing to developers, researchers, and educators who want to keep abreast of rapid technology change. With content that's accessible and useful today, this publication acts as a catalyst for progress in this emerging field, bringing together the leading experts in such areas as

- Hardware technologies
- Software infrastructure
- Sensing and interaction with the physical world
- Graceful integration of human users
- Systems considerations, including scalability, security, and privacy

Subscribe Now!

VISIT www.computer.org/pervasive