

Chapter 7

Avoid Over and Under Design

Developers tend to take one of two approaches to programming. Many think they need to plan ahead to ensure that their system can handle new requirements that come their way. Unfortunately, this planning ahead often involves adding code to handle situations that never come up. The end result is code that is more complex than it needs to be and therefore harder to change – the exact situation we were trying to avoid. The alternative, of course, seems equally bad. That is, just jump in, code with no forethought and hope for the best. But this hacking also typically results in code that is hard to modify. What are we supposed to do that doesn't cause extra complexity, but leaves our code easy to change. The middle ground can be summed up by something I heard Ward Cunningham say at a user group – “Take as much time as you need to make your code quality as high as it can be, but don't spend a second adding functionality that you don't need now!” In other words, write high quality code, but don't write extra code.

This chapter is admittedly more of a new mantra than it is a detailed description of a technique to implement. This chapter takes advantage of what we've learned in “Encapsulate That” and sets the groundwork for “Refactor to the Open-Closed.”

A Mantra for Development

We believe developers should have a particular attitude when writing code. There are actually several we've come up with over the time – all somewhat being consistent with each other, but saying things a different way. Here are the ones we've held to date:

- Avoid over and under design
- Minimize complexity and rework
- Never make your code worse (the Hippocratic code of coding)
- Only degrade your code intentionally
- Keep your code easy to change, robust, and safe to change

Before we can discuss these mantras, we need to be clear what we mean by quality code. The appendix contains a chapter from Alan Shalloway's and Jim Trott's *Design Patterns Explained: A New Perspective on Object Oriented Design*. We'll give a brief, summary of code quality here, but interested readers may want to read the more extensive narrative in the appendix.

The Pathologies of Code Qualities

It's often easier to see code qualities by discussing examples of when the qualities aren't present. Let's look at five common code qualities: cohesion, coupling, redundancy, readability, and encapsulation.

Cohesion. Strongly cohesive classes are classes whose functions are all related to each other. Strongly cohesive methods are methods that only do one thing. The pathology of strong cohesion is classes or methods that do unrelated things. We've heard very weakly cohesive classes called "god objects" presumably because they are somewhat omniscient in that everything takes place in them¹.

Proper coupling. Having well-defined relationships between objects makes them easier to understand and likely to inadvertently cause problems when changing code. The pathology of improper coupling is the occurrence of side effects – that is, unexpected errors due to making changes elsewhere.

No Redundancy. No redundancy is difficult to achieve. The more redundancy you have the more time it will take to make changes. As we discussed in the Shalloway's Law and Shalloway's Principle chapter, no redundancy is virtually impossible to achieve – but at least you want to make it so you don't have to find the duplication. Essentially the pathology of no redundancy is that when you make a change in one place you have to make a change in another place.

Readability. Readable code means you can understand what has been written. It requires intention revealing names and is best achieved by using programming by intention (see Chapter <<>> Programming by Intention). Unreadable code, of course, is code you can't understand when you read it. Poor names, tight coupling and big methods/classes contribute greatly to the unreadability of code.

Encapsulation. Encapsulation is more than mere data-hiding. The type of an object is one of the most important things to hide. Design patterns are really about hiding: object type, cardinality, which function is being used, order, optional behavior, construction, and more. The pathology of encapsulation is when you must know how the code you are using is implanted in order to use it properly. This often means you know the implementation type of the object being used or know something about cardinality, order, etc.

Avoid Over and Under Design

This essentially means put in the correct amount of design. Over design is putting in things that add complexity to the code that may or may not be needed. Note, the key word here is complexity. We're not as worried about the time you take as much as we are about how you leave the state of the code. If the work you've done does not raise the complexity of the code you have, then no worries. In other words, putting in an interface where one may or may not be needed is not necessarily a bad thing if everyone understands interfaces. Interfaces aren't really complexity adders in our mind. They are a holder for an idea. However, putting in a complex parameter list (or using a value object to hold a parameter, say, when one isn't needed) would be raising complexity.

Under design is actually a euphemism for "poor code quality." I would view under design as having taken place when high coupling and/or weak cohesion are present. Typically proper encapsulation is also not present. So, avoiding over design means make your code changeable but don't add things that you don't need now. If you need them later the changeability of the code will enable you to do that will list, if any, extra cost. Avoiding under design mostly means make sure your code is changeable.

¹ We've also thought they may be called this because when you first look at them you mutter to yourself – "oh my god!" and the fact that it looks like only god could figure them out.

Minimize complexity and rework

Many people only partly understand the true nature of refactoring. Martin Fowler, in his excellent “Refactoring: Improving the Design of Existing Code²” describes refactoring as

*“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.”**

In the book, Fowler talks about refactoring as a method of cleaning up messy/poor code. However, there is another side to refactoring that Fowler doesn’t talk about. This is refactoring code that is of high quality, when it comes to the code qualities we’ve been talking about, but which no longer has sufficient design due to new requirements. In other words, the book talks about how to clean up poorly written code (a good thing to know) but mostly ignores how to refactor good code that now must be changed to accommodate new requirements³.

We strongly suggest that refactoring good code when new requirements come so that the code is better able to accommodate the changes is a way to minimize complexity because you are deferring adding complexity until it is needed, but your code is high so there is no rework. We would contend that delaying extensions to code is not rework, but a kind of Just-In-Time design. We’ll talk explicitly about how to do this in our Refactoring to the Open-Closed chapter.

Never make your code worse / Only Degrade Your Code Intentionally

Existing code degrades one bit at a time (no pun intended). We suggest that team members do their best to not take shortcuts which makes their code worse. Sometimes this is difficult however. It may be that legacy code makes it very difficult to add functionality in properly without harming your code. To be realistic, we restate the “Never make your code worse” to “Only degrade your code intentionally.” While this may sound funny, the alternative would be to make your code worse *unintentionally*.

One way to only degrade your code intentionally is to ensure you consider alternatives. One way to do this is to make a team wide agreement that if a developer can’t figure out how to make a change without degrading code, he/she will tell another team member of the change he/she is thinking of making before they make the change. Note that we are not requiring getting permission or even getting a better result. We’re just suggesting you tell someone. This forces you to at least reflect a little. Our experience has shown us that people stop just short of good solutions because they are willing to take the first thing that comes to their mind. This forces them to think about things a bit more (sometimes a lot more because they don’t want to admit to a co-worker that they don’t have a good solution).

Keep Your Code Easy to Change, Robust and Safe to Change

Code should not be viscous. That means, that the effort to make changes should not be excessive. Viscosity can be avoided by having easy to understand, non-redundant code. Code should also not be brittle. That is, changes in one place should not break code in other places. This requires loosely coupled code, following Shalloway’s Principle (see Chapter <<>>

² Refactoring: Improving the Design of Existing Code, Martin Fowler, page <<<<>>>>

³ Alan Shalloway had a private conversation with Martin about this once. After suggesting that the refactoring concepts Martin presented would work equally well for both types of code, Martin responded by agreeing and saying “my book was long enough as it was!”

Shalloway's Law and Shalloway's Principle) and proper encapsulation. It is not sufficient to follow these two mantras alone, however. While doing so may make it easy to change your code with less likelihood of breaking it, there are no guarantees. The only way to be assured that you can safely change your code is to have a full set of automated acceptance tests available.

Summary

Developers must always be aware of doing too much or too little. When you anticipate what is needed and put functionality to handle it, you are very likely to be adding complexity that may not be needed. If you don't pay attention to your code quality, however, you are setting yourself up for rework and problems later. Code quality is a guide. Design patterns can help you maintain it because they give you examples of how others have solved the problem in the past in similar situations.

Business-Driven Software Development (BDSD) is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDSB goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSB integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. In BDSB:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality

Become a Lean-Agile Enterprise

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSB can bridge this gap by providing the practices for managing the flow of work.

Why Net Objectives

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

<p>Assessments</p> <p>See where you are, where you want to go, and how to get there.</p> <p>Business and Management Training</p> <p>Lean Software Development Product Portfolio Management Enterprise Release Planning</p>	<p>Productive Lean-Agile Team Training</p> <p>Team training in Kanban, Scrum Technical Training in ATDD, TDD, Design Patterns</p> <p>Roles Training</p> <p>Lean-Agile Project Manager Product Owner</p>
--	---

