Chapter 13. Software Architecture and Design's Role in Lean-Agile Software Development

"If builders built buildings the way programmers wrote programs, the first woodpecker that came along would destroy civilization." —Gerald Weinberg

"Prediction is very difficult, especially about the future." —Niels Bohr

IN THIS CHAPTER

At the beginning of the Agile movement, software architecture and design was often considered irrelevant. Most Agile advocates have since recanted this notion; however, the damage still lingers. In Agile development, proper software architecture and design is still not well understood. This chapter discusses the changing role of software architecture and design, from a framework that holds pieces together to a framework that enables change as requirements evolve. We also discuss the need for design patterns and test-driven development.

Software architecture and design is somewhat of a technical issue and this is not a technical book. Nevertheless, it is important to touch on the topic because it is an essential aspect of Lean-Agile software development. There are a number of excellent books that cover software design in more detail, such as *Design Patterns Explained: A New Perspective on Object-Oriented Design* (Shalloway and Trott 2004), *Emergent Design: The Evolutionary Nature of Professional Software Development* (Bain 2008), *Agile Software Development: Principles, Patterns and Practices*

(Martin 2002), and the forthcoming *Essential Skills for the Agile Developer: A Guide to Better Programming and Design* (Shalloway and Bain 2010).

Note: In this chapter, we will refer to software architecture and design simply as "software design."

Takeaways

Key insights from this chapter include

• Software design is not static; it must evolve as our understanding evolves.

• Developers must attend to both design quality and automated testing in order to provide software systems that can change quickly when needed.

AVOIDING OVER- AND UNDER-DESIGN

None of us is very good at anticipating future needs. [1] This leads to a common concern among developers: how to avoid over-designing and under-designing the system. That is, while you know that you don't want to overbuild your system, you don't want to be hacking in your solutions either. The remedy is

1. Alan likes to say that "we are all pre-cognitively impaired." Every one of us is limited in our ability to predict the future.

Build only what you need at the moment and build it in a way that allows for it to be changed readily as you discover new issues.

Most developers have had an experience in which they were given some unexpected requirements that were difficult for the system to accommodate. The pain of going through this once or twice leads most

of us to try to anticipate what will be needed in the future, which invariably leads to building more than is truly necessary. That leads to systems that are more complex than they need to be. And over the long term, complexity slows teams down.

To handle this problem, developers need to be able to do the following:

• Write code quickly.

• Make changes to code without breaking it.

• Be able to change code safely; that is, if you break it, know you've broken it.

Doing this enables developers to add functionality to systems in a fast, safe, efficient manner. Following are three questions you can use to self-assess whether your team can safely and efficiently change code.

• Can you easily change your code?

• If you change it, are you likely to break it?

• If you break it, can you automatically detect the break?

People who are new to Agile might think that Agile compounds the problem of modifying code because it is geared toward making quick changes. But in truth, changing code has become a way of life no matter how you do development. Requirements come at development teams much faster than they used to. The speed of change in the software industry has accelerated so much that a continuously evolving system is a way of life. If we aren't improving our system continuously, we are already falling behind.

Agile highlights the problem because of its emphasis on effectiveness. It has become common practice in Agile approaches to require automated regression tests so as to detect the consequences of changes. This is good practice for safety and for efficiency.

But automated acceptance tests are only part of the answer to changing code quickly. Accommodating change also requires good design. This is why design patterns have become an essential part of any competent developer's toolbox. Unfortunately, design patterns are little understood (or, more correctly, largely misunderstood). [2]

2 . Improving legacy code to handle change is also essential to consider but that is beyond the scope of this book. *Working Effectively with Legacy Code* (Feathers 2004) provides an excellent treatment of this important topic.

From what we've seen in working with dozens of clients, we fear our industry does not measure up to the test. We say this because of the almost universal answer we get to the following question:

Imagine you are working on a well established system and you need to add some new functionality. Where will you spend most of your time—in writing the new functionality or in integrating the functionality into your existing system?

For the past ten years, we've been asking this question; consistently—95 percent of the time—the answer is, "integrating it in."

Most organizations will not pass this test. By and large, developers do not write their code to be changeable. Instead, they focus on implementing the task at hand and don't recognize when they should put in a design layer, or when they shouldn't. Putting layers in your code whenever possible is a path to overly complex code; but not putting in layers when you need to is a path to code that is both difficult to change and brittle.

DESIGNING FOR CHANGE

So how do you build for change?

One approach is to use the simple question, "How would I design this if I found out later that however I designed it now was not the best

way?" Our experience in design-patterns training leads us to believe that most developers do not like to think abstractly. Combine that with the pressure they are under and it isn't surprising that often they just deal with the task at hand instead of reflecting on what a general solution might be. Given time to reflect, they unfortunately often go the other way—and overbuild to handle anything that might come up. The trick is to realize at the beginning that it is unlikely you'll make the right decision—that is, you must write your code so that it is able to handle change, but it is unlikely you'll know how it will change. There is too much unknown and too much to learn—not only by the developers, but also by the users or whoever is speaking for them. As the system progresses through the development process, more ideas will come to the fore. The key is to write high-quality code so that the system is changeable, and full acceptance tests so that it is *safely* changeable. Management needs to support and encourage teams to do this.

This attitude is based on a characteristic of programming that most developers have come to understand:

Other than for exceptionally complex functions, more effort is typically required in handling the coupling of a function to other elements in the code than in writing the function itself.

DOING JUST ENOUGH UP FRONT

A (fortunately dying) myth in Agile software is that it is bad to do any sort of design work up front. While Agile grew up partly as a counter to over-design (big design up front—BDUF), no design is going too far. From a design point of view, what we need early on is the big picture so that we can identify the main concepts in the problem domain and determine how they relate to each other. This gives us "just enough" detail to create a conceptual framework within which to think of the problem domain. Then, as we become

aware of new concepts, we can see how to add them, how they fit in. In contrast, BDUF can actually obscure the big-picture view by giving us too many details and too much complexity.

If you are interested in how to create a high-level design that identifies the concepts in a problem domain, see the chapters on Commonality and Variability Analysis and the Analysis Matrix in *Design Patterns Explained: A New Perspective on Object-Oriented Design* by two of this book's authors (Shalloway and Trott). These chapters cover some of the basics of identifying the essential application architecture that should be discovered prior to actually writing code.

THE ROLE OF DESIGN IN SOFTWARE

The role of design in software is to make it easy to change code, to minimize the effect of changes in the system. This can be done through a combination of decoupling (isolation), encapsulation, and avoiding redundancy. In other words, the purpose of software architecture is not to define a place for each of the pieces as much as it is to handle properly the dependencies of the pieces.

Software design needs to evolve as more is learned about what the program is supposed to do. We inject better designs as needed, following what we call "Just-in-Time Design." This avoids over-building designs in anticipation of what may be needed later, because developers often over-anticipate what is needed.

THE ROLE OF MANAGEMENT IN SOFTWARE DESIGN

Management's role is mostly to support the software development team while providing a vision of what needs to be built. Part of this support is to help the team focus on what they need to do without overly pressuring them to cut corners, especially when it comes to creating automated tests. This does not mean that management should

just quietly accept everything development teams want to do. Management needs to ensure that there is a cost justification for development's efforts. However, they should trust the development team's judgment when it comes to how to build quality.

SUMMARY

The purpose of software design is not to build a framework within which all things can fit nicely. It is to define the relationships between the major concepts of the system so that when they change or new requirements emerge, the impact of the changes required is limited to local modifications.

TRY THIS

These exercises are best done as a conversation with someone in your organization. After each exercise, ask each other if there are any actions either of you can take to improve your situation.

• As a manager:

• Ask yourself, "What is the business value of having a flexible system?

• Do you trust your team to build only what they need?

• Why or why not?

• As a developer:

• Ask yourself what basis you use for deciding to improve the infrastructure of your system.

• Do you feel supported by management to build in the right amount of quality to your software?

• Do you feel management understands the value of quality in software?

RECOMMENDED READING