Designing with OOP

CHAPTER 10

> *That's right!" shouted Vroomfondel. "We demand rigidly defined areas of doubt and uncertainty!"*
>
> *The* Hitchhiker's Guide to the Galaxy*, by Douglas Adams*

What is design? How do you go about designing with objects? As far as a CPU is concerned, you don't need objects. You might as well put all the code into one big function—but that code would be really, really hard to understand. So we break up a single huge solution into a number of smaller, understandable, and rigidly defined pieces. This is the essence of design, regardless of the language.

When we design with OOP, this "breaking up" is really just the construction of a network of classes. However, before we dive into making classes, we need to understand the OOP design bias. OOP designs tend to focus on roles and responsibilities. These roles and responsibilities get broken down further into smaller networks of classes. Ultimately, we wiggle wires to communicate with the chip.

In this chapter we introduce some basic guidelines for design. We also talk a bit about some common design mistakes and how to avoid them.

Overview

Design is so intertwined with coding that it's artificial to separate the two. Academic textbooks explain that first you architect, then you design, and then you code. In the real world, however, these steps all get jumbled together. We tend to do all three at once, having a general idea of what we want to do, then refining and changing our idea as we start to code.

Designing with OOP is no different. We talk about "paper napkin" or whiteboard designs. We prototype and refine class interface files and talk about what each class should do. Just as important, we talk about how the classes interact and exchange control and data.

Design is messy, but designing with classes can be a little cleaner. This chapter provides some general guidelines that can help with this inherently untidy process.

Keeping the Abstraction Level Consistent

A key evolution in programming came about when we started to talk about "abstraction levels" in a design. This is somewhat expected, because humans are abstraction machines. A child can recognize a "chair," from the folding chairs at school to the hydraulic ones we use at work, and most people can operate a car, regardless of the make or model. Our mind's ability to abstract away the details of an object or process is directly applicable to programming. We can solve a complex design by using abstractions, from the big-picture operations at the top, down to the wire protocols at the chip interface level.

To put this in fancier terms, at any layer in a design there is an associated *scope of concern* and an appropriate *level of detail*. A scope of concern is the role of the task. The level of detail is the responsibility of the task.

At the top level of the verification system, the scope of concern is the entire chip, its configurations, and the traffic that will be applied to the chip in testing or real life. Here, the level of detail should be very small. In other words, the test should consist of "big" objects, such as the test and testbench, and have no minutia. At the other end of the spectrum, at