Thinking OOP

CHAPTER 9

NOBODY expects the Spanish Inquisition! Amongst our weaponry are such diverse elements as fear, surprise, ruthless efficiency, an almost fanatical devotion to the Pope, and nice red uniforms—Oh damn!

Monty Python, episode 15, 1970

Getting your brain around OOP is a challenge. You may have followed the syntax of classes, inheritance, and so on. But when should you write classes or use inheritance? What about templating and operator overloading? A little befuddlement is okay—OOP requires a shift in thinking, and mental fog is a natural result.

This chapter will get you "thinking OOP." The reason OOP is all muddy is that there are no rules. "Thinking OOP" is more about using a set of coding biases and lessons learned than in making trade-offs. Sure, we could have pretended there were rules, providing numbered steps such as, "first you must blah, blah, blah," or "you must always apply by blah, blah, blah," but no one would remember. Instead, this handbook tries to teach you how to ride the "OOP bicycle." Learning to "think OOP" is not trivial, but once you've learned, you never forget.

Overview

We now introduce thinking and using OOP in stages. From the first stage (the "big picture") to the last (coding), we introduce an "arsenal of weaponry" that has proved useful for programmers. This arsenal requires a few chapters. In this chapter we concentrate on framing the OOP process. We talk about the difficulties in managing complexity and creating adaptable code. We then discuss the difference between the interface and the implementation of a piece of code. Subsequent chapters cover architecture and coding.

Remember, verification is neither simple nor easy. Any serious attempt to verify hardware will result in a complex system. Consequently, it is important to realize that the complexity of a verification system is not the result of poor implementation, but is largely intrinsic to the problem of verifying a complex design.

Object-oriented programming in C++ was developed to help manage complex problems,[1] not eliminate them. The goal is to make the complex appear simple without introducing unexpected behavior. The trick is to keep the focus on making things seem as simple and clear as possible, while minimizing the use of "magic" code or confusing connections. This will nonetheless create a bit of a conundrum, as what is simple and clean to one is often perceived as unnecessarily complex and "sneaky" by another.

There is no "silver bullet" to slay the werewolf of complexity. Verification complexity needs to be managed differently across different types of projects. For example, System-on-a-Chip (SoC) designs are complex because they often involve several independent input/output (I/O) subsystems. SoC and graphics chip designs are complex because of pipelined and interrelated processing. The best solution to complexity is communication, through understandable design and code (abstractions, minimal assumptions, and so on), combined with a drive toward common-sense simplicity.

[1] In an ironic twist of fate, "C with Classes," the progenitor of C++, was invented to solve simulation problems.