Unit Testing Antipatterns
(The “Don’ts”)

The majority of software projects are ongoing affairs. New screens and functions are released on (or around) specific, agreed dates, but a project itself keeps chugging along. So it’s far more common for developers to join a project mid-flow, with its own dusty corners that no one dares to disturb, its homegrown object-relational mapping classes, crusted together with the foibles and wrinkles of a dozen long-since departed self-styled code gurus. Starting a brand new project from scratch is a rare joy: the opportunity to set the standards, consider the architecture, evaluate the available technologies given the business requirements, and produce clean, maintainable code that’s easily unit-tested.
Code that’s difficult to test tends to also be poorly designed code: difficult to maintain, a nightmare to debug, time-consuming, and obstinate, when all you want to do is add a new field, track down a particular database column reference, or figure out a calculation that snakes from an asynchronous listener object to a UI component and back.

Probably the best way to illustrate what’s needed from a good design is to start by looking at a really bad design, and examine why it’s bad—or in this case, why it’s a nightmare to write tests for. That’s what we’ll do in this chapter. The idea of this chapter and the next is to provide a set of design criteria to think about while doing detailed design, and while turning the design into code and tests.

Caution All of the “testing antipatterns” described in this chapter are based on the point of view of improving testability. There may well be other concerns that trump this particular concern—e.g., in massively parallel, high-performance computing applications, factors such as execution efficiency and memory usage have to trump class-level encapsulation. As with most considerations in software design, it’s a case of weighing up what’s most important for this project, and producing a design to match.

The Temple of Doom (aka The Code)

We’ve built this entire chapter around a single code example. We refer to that example as The Temple of Doom, because in it we’ve consolidated what we believe to be the ten worst antipatterns (or suicidal practices) that make unit testing really difficult. We’ve marked each of our suicidal practices with an image of Ixtab, the Mayan goddess of suicide:

1. Half-baked features from the Good Deed Coder
2. Service objects that are declared final
3. Privates on parade
4. Business logic in the UI code
5. The tightly bound dependency
6. The Singleton design pattern
7. Static methods and variables
8. The static hair-trigger
9. The stratospheric class hierarchy
10. The complex constructor

All of these (very common) code constructs make your code difficult to unit-test. We’re going to present a Java class implementing all ten at once, so brace yourselves for the big blob of bloated bleariness.