In this chapter we emerge from the Temple of Doom to find the sun shining brightly and the water a lovely shade of turquoise...

If you’ve just finished reading Chapter 9, you should now be feeling utterly deflated and at a loss for the sanity of the programming world, to the point where even standing up to make a cup of tea feels like too much effort. So let us lift your spirits with the inverse of the Temple of Doom code base from Chapter 9. In this chapter we’ll sweep away the difficult-to-test code and start over with a solid use case, a design, and a clear goal for the finished product. We hope you appreciate the difference, and

1 You may have to take our word for the turquoise part... or, alternatively, check out the original image, along with more of Doug’s photography from Tulum and Cancun (including 360-degree virtual tours), at www.vresorts.com/VRPages/Cancun/Cancun.html.
that within a few pages you'll feel sufficiently uplifted to make that cup of tea. (Whoever said that software development isn’t an emotional rollercoaster ride?)

In a moment we’ll present the top ten “Design For Testing” to-do list; the bulk of the chapter will be structured around these (just as Chapter 9 was structured around the “don’ts”). However—and this is really the key to easier testing—before we get started on the design, we’ll start with a use case. By having a clear idea of what the finished product is meant to do, we’ll know exactly what we want to achieve from the design. So if there’s any doubt about whether a feature or design element should be included, we can ask: “Is it within the scope of our use case?”

Top Ten “Design for Testing” To-Do List

The following list is really the flipside of the top ten unit testing antipatterns (or “don’ts”) from Chapter 9. If you keep the following design criteria in mind while creating a detailed design (see the next chapter), your design will be more amenable to unit testing (and also more pliant, easier to understand and maintain, and more likely to be picked up and used while the code is being cut):

10. **Keep initialization code out of the constructor.**
   Instead, create a separate `initialize()` method, to allow the object to be fully constructed before it starts strutting its stuff.

9. **Use inheritance sparingly.**
   Use inheritance sparingly, and only if it absolutely makes sense to use it. Instead, as a general guideline (but not an all-out rule) use aggregation instead. Use inheritance when you are adding behavior that directly manipulates private data and can do so with minimal dependence on other objects; use aggregation otherwise.

8. **Avoid using static initializer blocks.**
   A static initialization block is a block of code that gets run the moment a class is loaded into memory—so the code will be run the first time a class is referenced, even in an unassigned variable, by some other class. This is why we refer to these static initialization blocks as “static hair triggers.”

7. **Use object-level methods and variables instead of static (class-level) methods and variables.**
   This rule holds true unless the variable in question is immutable and (if it’s a complex object) has immutable data and little or no construction code.

6. **Use a Registry or some other pattern instead of the Singleton design pattern.**
   While singletons themselves are sometimes necessary, the Singleton design pattern itself has its problems, not least of which is that it makes unit testing difficult. Instead use a Registry class, or whatever mechanism fits your project, to access single instances. Also, question the need for a singleton: does the object in question really need to be the only instance in the system?

5. **Keep your classes as decoupled as the application permits.**
   Avoid tying your code too tightly to dependent implementations. Instead, allow callers to pass in their own implementation. In other words, “future-proof” your code (and make testing easier) by keeping it free of assumptions about how it’ll be used. (That said, sometimes there are legitimate object collaborations driven by the application functionality, e.g., where a class delegates core functions that more properly belong to it.)