Hard-to-Test Code

Code is difficult to test.

Automated testing is a powerful tool that helps us develop software quickly even when we have a large code base to maintain. Of course, it provides these benefits only if most of our code is protected by Fully Automated Tests (see page 26). The effort of writing these tests must be added to the effort of writing the product code they verify. Not surprisingly, we would prefer to make it easy to write the automated tests.[3]

[3] We would also like to recoup this cost by reducing effort somewhere else. The best way to achieve this is to avoid Frequent Debugging (page 248) by writing the tests first and achieving Defect Localization (see page 22).

Hard-to-Test Code is one factor that makes it difficult to write complete, correct automated tests in a cost-efficient manner.

Symptoms

Some kinds of code are inherently difficult to test—GUI components, multithreaded code, and test code, for example. It may be difficult to get at the code to be tested because it is not visible to a test. It may be problematic to compile a test because the code is too highly coupled to other classes. It may be hard to create an instance of the object because the constructors don't exist, are private, or take too many other objects as parameters.

Impact

Whenever we have Hard-to-Test Code, we cannot easily verify the quality of that code in an automated way. While manual quality assessment is often possible, it doesn't scale very well because the effort to perform this assessment after each code change usually means it doesn't get done. Nor is this strategy readily repeated without a large test documentation cost.

Solution Patterns

A better solution is to make the code more amenable to testing. This topic is big enough that it warrants a whole chapter of its own, but this section covers a few of the highlights.

Causes

Also known as

Hard-Coded Dependency

There are a number of reasons for Hard-to-Test Code; the most common causes are discussed here.

Cause: Highly Coupled Code

Symptoms

A class cannot be tested without also testing several other classes.

Impact

Code that is highly coupled to other code is very difficult to unit test because it won't execute in isolation.

Root Cause

Highly Coupled Code can be caused by many factors, including poor design, lack of object-oriented design experience, and lack of a reward structure that encourages decoupling.

Possible Solution

The key to testing overly coupled code is to break the coupling. This happens naturally when we are doing test-driven development.

A technique that we often use to decouple code for the purpose of testing is a Test Double (page 522) or, more specifically, a Test Stub (page 529) or Mock Object (page 544). This topic is covered in much more detail in Chapter 11, Using Test Doubles.

Retrofitting tests onto existing code is a more challenging task, especially when we are dealing with a legacy code base. This is a big enough topic that Michael Feathers wrote a whole book on techniques for doing this, titled Working Effectively with Legacy Code [WEwLC].

Cause: Asynchronous Code

Symptoms

A class cannot be tested via direct method calls. The test must start an executable (such as a thread, process, or application) and wait until its start-up has finished before interacting with the executable.

Impact

Code that has an asynchronous interface is hard to test because the tests of these elements must coordinate their execution with that of the SUT. This requirement can add a lot of complexity to the tests and causes them to take much, much longer to run. The latter issue is a major concern with unit tests, which must run very quickly to ensure that developers will run them frequently.

Root Cause

The code that implements the algorithm we wish to test is highly coupled to the active object in which it normally executes.

Possible Solution

The key to testing asynchronous code is to separate the logic from the asynchronous access mechanism. The design-for-testability pattern Humble Object (page 695; including Humble Dialog and Humble Executable) is a good example of a way to restructure otherwise asynchronous code so it can be tested in a synchronous manner.

Cause: Untestable Test Code

Symptoms

The body of a Test Method (page 348) is obscure enough (Obscure Test; see page 186) or contains enough Conditional Test Logic (page 200) that we wonder whether the test is correct.

Impact

Any Conditional Test Logic within a Test Method has a higher probability of producing Buggy Tests (page 260) and will likely result in High Test Maintenance Cost (page 265). Too much code in the test method body can make the test hard to understand and hard to construct correctly.

Root Cause

The code within the body of the Test Method is inherently hard to test using a Self-Checking Test (see page 26). To do so, we would have to replace the SUT with a Test Double that injects the target error and then run the test method inside another Expected Exception Test (see Test Method) method—much too much trouble to bother with in all but the most unusual circumstances.

Possible Solution

We can remove the need to test the body of a Test Method by making it extremely simple and relocating any Conditional Test Logic from it into Test Utility Methods (page 599), for which we can easily write Self-Checking Tests.