Appendix A: Writing Testable Code

The Xcode integrated support for testing makes it possible for you to write tests to support your development efforts in a variety of ways. You can use tests to detect potential regressions in your code, to spot the expected successes and failures, and to validate the behavior of your app. Testing improves the stability of your code by ensuring that objects behave in the expected ways.

Of course, the level of stability you achieve through testing depends on the quality of the tests you write. Likewise, the ease of writing good tests depends on your approach to writing code. Writing code that is designed for testing helps ensure that you write good tests. Read the following guidelines to ensure that your code is testable and to ease the process of writing good tests.

Guidelines

- **Define API requirements.** It is important to define requirements and outcomes for each method or function that you add to your project. For requirements, include input and output ranges, exceptions thrown and the conditions under which they are raised, and the type of values returned (especially if the values are instances of classes). Specifying requirements and making sure that requirements are met in your code help you write robust, secure code.

  See the Unit Testing Apps and Frameworks sample-code project for an example of using exceptions to identify and report incorrect library usage by client code.
- **Write test cases as you write code.** As you design and write each method or function, write one or more test cases to ensure that the API's requirements are met. Remember that it's harder to write tests for existing code than for code you are writing.
- **Check boundary conditions.** If a parameter for a method must have values in a specific range, your tests should pass values that include the lowest and highest values of the range. For example, if a procedure has an integer parameter that can have values between `0` and `100`, inclusive, the test code for that method should pass the values `0`, `50`, and `100` for the parameter.
- **Use negative tests.** Negative tests ensure that your code responds to error conditions appropriately. Verify that your code behaves correctly when it receives invalid or unexpected input values. Also verify that it returns error codes or raises exceptions when it should. For example, if an integer parameter must have values in the range `0` to `100`, inclusive, create test cases that pass the values `-1` and `101` to ensure that the procedure raises an exception or returns an error code.
- **Write comprehensive test cases.** Comprehensive tests combine different code modules to implement some of the more complex behavior of your API. Although simple, isolated tests provide value, stacked tests exercise complex behaviors and tend to catch many more problems. These kinds of tests mimic the behavior of your code under more realistic conditions. For example, in addition to adding objects to an array, you could create the array, add several objects to it, remove a few of them using different methods, and then ensure that the set and number of remaining objects are correct.
- **Cover your bug fixes with test cases.** Whenever you fix a bug, write one or more tests cases that verify the fix.