Unit Testing

It’s as fun as it sounds!

Aaron DeGrow

Terminology

- Test Case
  - Exercises a unit of code and evaluates execution or performance
- Test Suite
  - A group of related test cases
- Test Specification
  - The test requirements (i.e., description of the test cases)
- Test Plan
  - The plan for testing (what should be tested, using what environment, etc.)

Functional Specification(s)

Really helpful for development and testing, but are often missing or lacking details.

- Specifies the functions and behavior of a system or component. Also known as the requirements
- Also a form of documentation on deliverables, expectations, etc.
- Extreme clarity on expected behaviors, making test development much easier
- Basis for "black box" test cases
- Allows test creation to happen in parallel to development

Designing Tests

- Unit tests should be simple and easy to write
- Each test should be constrained to testing one behavior/requirement
  - More than 1 assert in a test case may be a sign of testing too many things
- The number of test cases is often the result of the variations of the inputs and potential output of the code under test
- Not all variations need to be tested; only enough for sufficient coverage to thoroughly test the code
- Tests should be independent from each other and from external factors (no network, database, files, test/input data, etc.)
- UI tests are essentially integration tests

Parts of a Test

- Parts
  - Precondition(s)
  - Stimulus - The actual unit under test (i.e., the 'trigger' action)
  - Assertion(s)
    - Test expectations
    - Ideally from requirements or interface contracts (not from actual code behavior or developer guesses)
    - Use descriptive messages in asserts - include the 'what' and 'why'
- Alternatively called "given, when, then"
- Should be divided/indicated by whitespace or inline comments

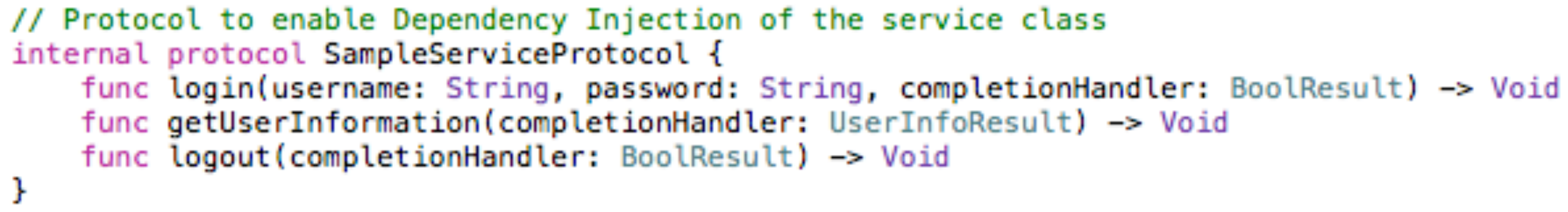
Organization & Naming

- Group tests primarily by class/feature/function
- Secondarily by:
  - Happy path/normal usage vs. boundary/bad input/error cases (expected vs unexpected)
  - Short vs. long running
  - Unit test vs. integration test
- Names should be clear to anyone (including yourself a year later) what's being tested
  - Naming convention for test cases should be similar to code style guide
  - Depending on the environment, usually must start with “test”
  - E.g., testFunctionNameBuildsMyObject()

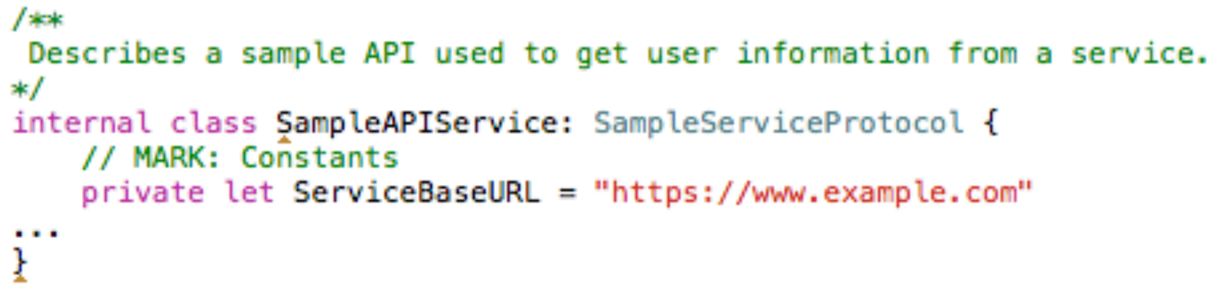
Mocking

- Simulated objects that mimic a real object in a controllable way to enable testing
- Particularly useful to simulate external interactions: network, database, file system, etc.
- "Flatten" async tests when possible
  - Tests that must wait for callbacks slow down testing and complicates test writing
  - To flatten:
    - Add properties/methods to the mock that allow you to specify the result/error response to a method before it's called
    - Have the mocked method call the callback method or block immediately with the pre-specified result/error response, then return
- Do not verify the behavior of (mock) dependencies (how they were interacted with) unless it is critical to the correctness of the code being tested

API Service That Conforms to a Protocol

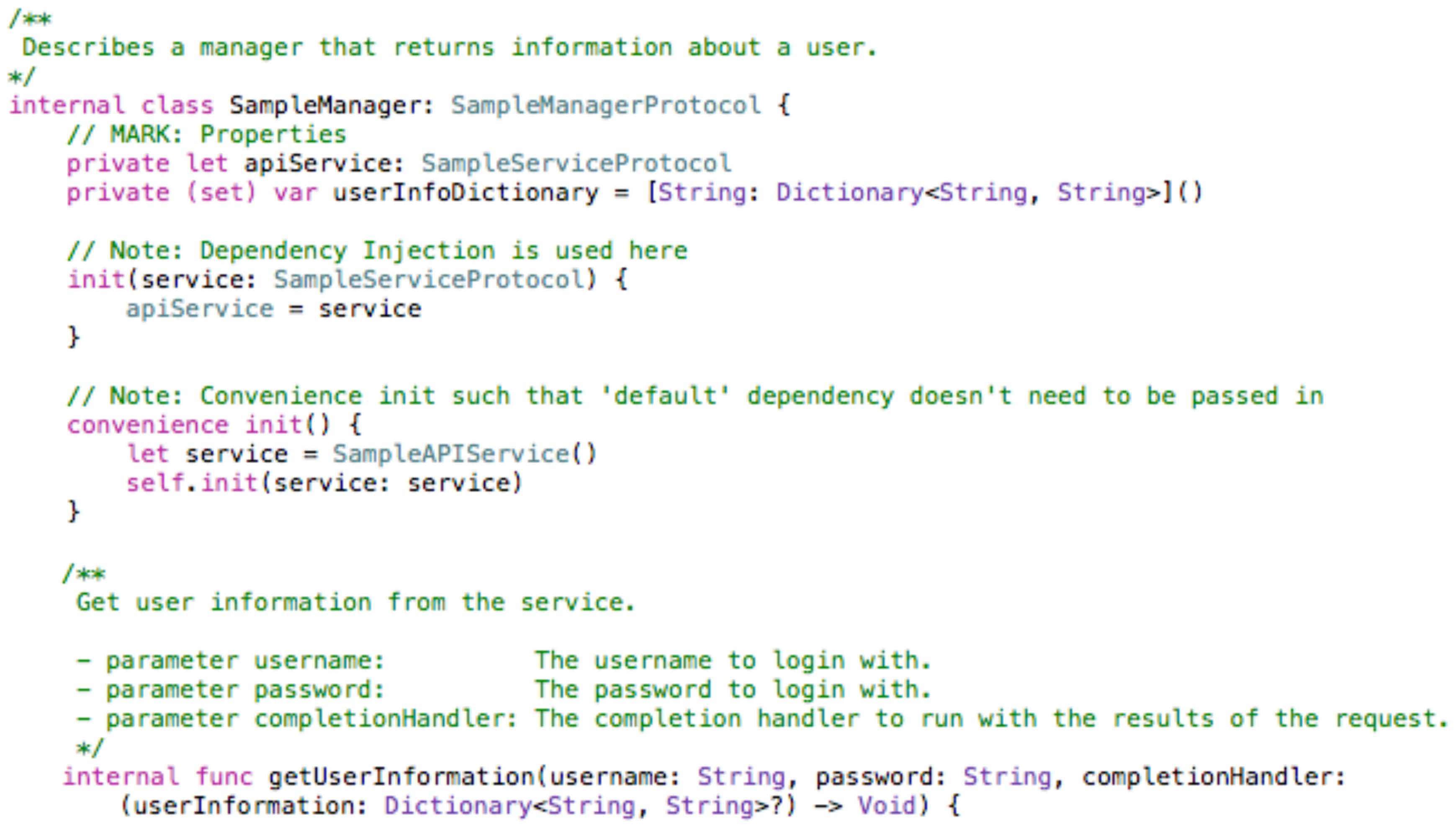


```swift
// Protocol to enable Dependency Injection of the service class
internal protocol SampleServiceProtocol {
    func login(username: String, password: String, completionHandler: BoolResult) -> Void
    func getUserInformation(completionHandler: UserInfoResult) -> Void
    func logout(completionHandler: BoolResult) -> Void
}
```



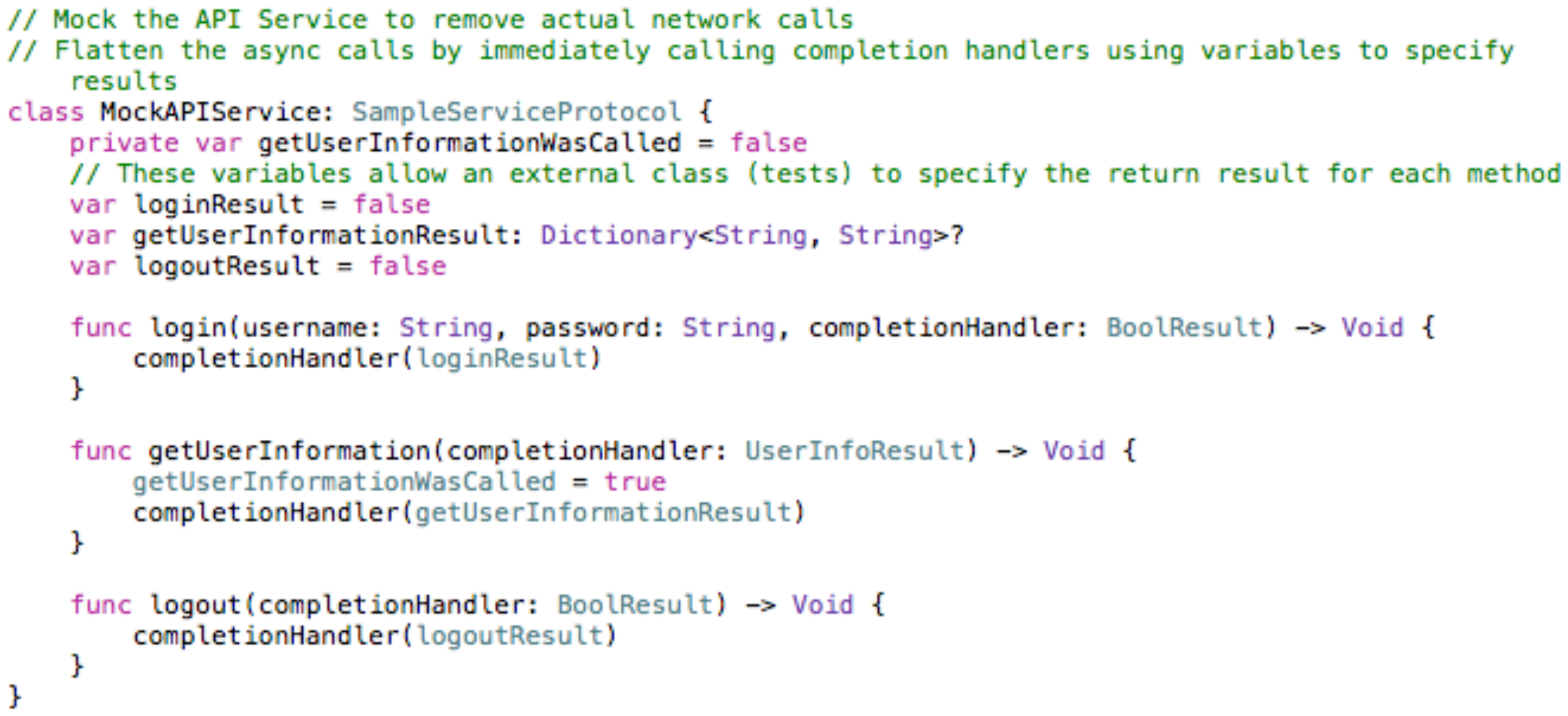
```swift
/**
 Describes a sample API used to get user information from a service.
*/
internal class SampleAPIService: SampleServiceProtocol {
    // MARK: Constants
    private let ServiceBaseURL = "https://www.example.com"
...
}
```

Manager That Uses the Service



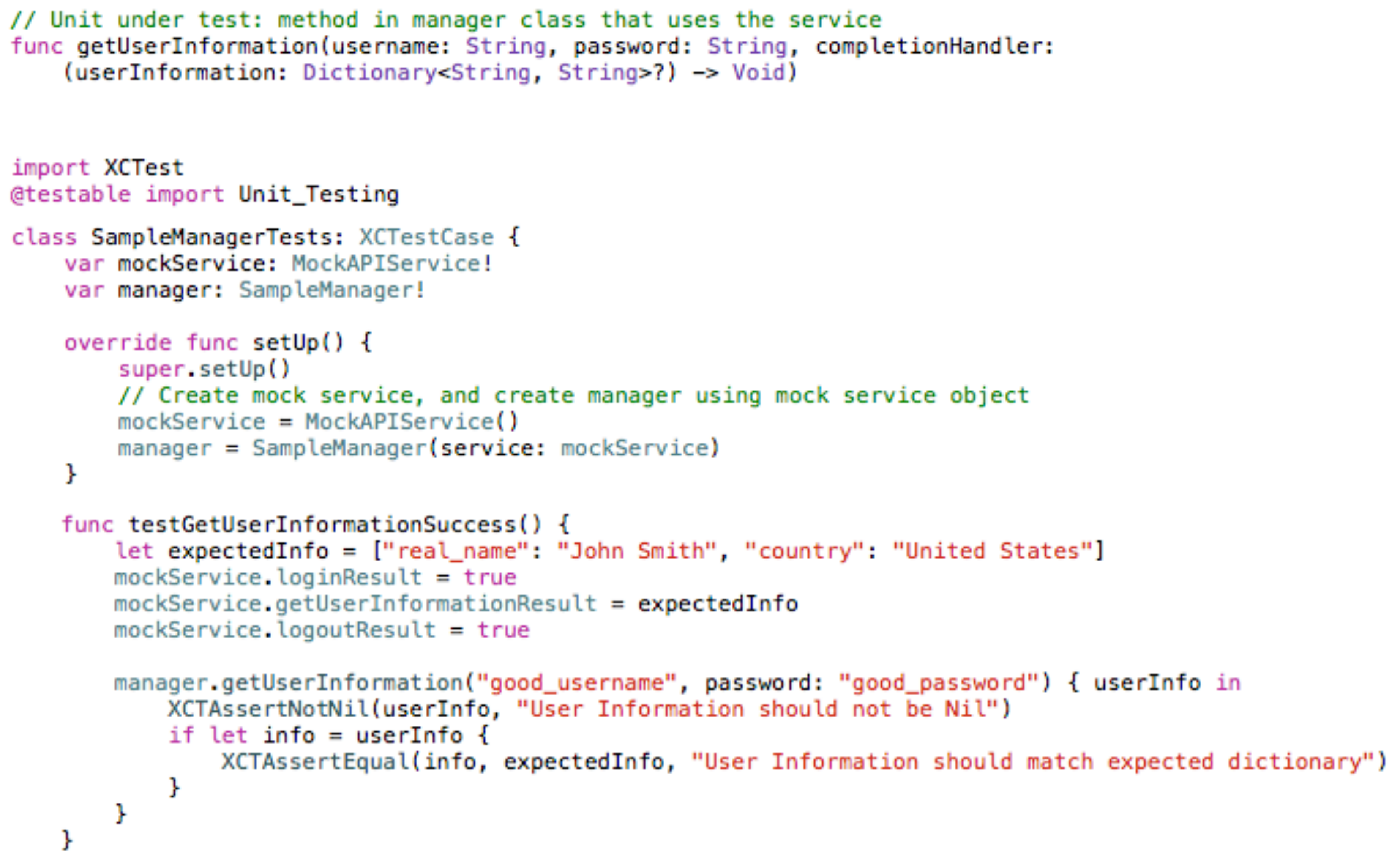
```swift
/**
 Describes a manager that returns information about a user.
*/
internal class SampleManager: SampleManagerProtocol {
    // MARK: Properties
    private let apiService: SampleServiceProtocol
    private (set) var userInfoDictionary = [String: Dictionary<String, String>]()

    // Note: Dependency Injection is used here
    init(service: SampleServiceProtocol) {
        apiService = service
    }

    // Note: Convenience init such that 'default' dependency doesn't need to be passed in
    convenience init() {
        let service = SampleAPIService()
        self.init(service: service)
    }

    /**
     Get user information from the service.

     - parameter username:          The username to login with.
     - parameter password:          The password to login with.
     - parameter completionHandler: The completion handler to run with the results of the request.
     */
    internal func getUserInformation(username: String, password: String, completionHandler:
        (userInformation: Dictionary<String, String>?) -> Void) {
```

Mock of the API Service



```swift
// Mock the API Service to remove actual network calls
// Flatten the async calls by immediately calling completion handlers using variables to specify
    results
class MockAPIService: SampleServiceProtocol {
    private var getUserInformationWasCalled = false
    // These variables allow an external class (tests) to specify the return result for each method
    var loginResult = false
    var getUserInformationResult: Dictionary<String, String>?
    var logoutResult = false

    func login(username: String, password: String, completionHandler: BoolResult) -> Void {
        completionHandler(loginResult)
    }

    func getUserInformation(completionHandler: UserInfoResult) -> Void {
        getUserInformationWasCalled = true
        completionHandler(getUserInformationResult)
    }

    func logout(completionHandler: BoolResult) -> Void {
        completionHandler(logoutResult)
    }
}
```

Sample Unit Test



```swift
// Unit under test: method in manager class that uses the service
func getUserInformation(username: String, password: String, completionHandler:
    (userInformation: Dictionary<String, String>?) -> Void)


import XCTest
@testable import Unit_Testing

class SampleManagerTests: XCTestCase {
    var mockService: MockAPIService!
    var manager: SampleManager!

    override func setUp() {
        super.setUp()
        // Create mock service, and create manager using mock service object
        mockService = MockAPIService()
        manager = SampleManager(service: mockService)
    }

    func testGetUserInformationSuccess() {
        let expectedInfo = ["real_name": "John Smith", "country": "United States"]
        mockService.loginResult = true
        mockService.getUserInformationResult = expectedInfo
        mockService.logoutResult = true

        manager.getUserInformation("good_username", password: "good_password") { userInfo in
            XCTAssertNotNil(userInfo, "User Information should not be Nil")
            if let info = userInfo {
                XCTAssertEqual(info, expectedInfo, "User Information should match expected dictionary")
            }
        }
    }
```

Test Friendly Design

- Architecture can strongly affect testability
  - Abstractions, Separation of Concerns, etc.
- Use Protocol(Interface)-Oriented Programming
  - Swift was designed to be a protocol-oriented language
  - Makes Dependency Injection easier
  - https://developer.apple.com/videos/play/wwdc2015/408/
- Use Dependency Injection (D.I.)
  - Makes the use of mocks much easier
- Reasons to use D.I. for mocking (instead of subclassing):
  - If you don’t override everything in a subclass, unintended behavior is possible
  - Subclasses are unstable with built-in (framework/library) classes
    - You’d have to update the subclass whenever new methods are added (to override them)

Best Practices

- Tests should be fast and yield the same result each iteration
- Tests should perform all evaluation/validation on their own (no human evaluation needed)
- Don't tie tests to implementation details (I.e., Black-box tests preferred)
- Input data should come from the test itself, not an external environment/source
  - External resources are problematic: Can’t be controlled easily, slow, can fail
- Don't include unnecessary assertions
- Avoid unnecessary preconditions
- Minimize common setup/teardown code

Best Practices 2

- Don't test private methods unless there's a very good reason to
- Each test should add value, not just inflate the code base
- 3rd party code should never be mentioned in tests
  - Libraries could change their API or be swapped out entirely
  - Treat them as a dependency and use D.I. w/ a protocol
- Constructors should not be tested (and shouldn't have code that needs testing)
- When a bug is found in QA or production, add a new unit test for it if applicable
- Code coverage metrics/tools can be unreliable
  - Instead, focus on testing the different states of the code

Miscellaneous

- Global state is hard to test; avoid it if possible
  - Static properties, Singletons, etc. => Bad
- If a test is hard to write it is an indication that the unit (or interface) being tested may need redesign.
  - E.g., Hard to get into the right state, hard to specify expectation, etc.

iOS Specific

- Use XCTest framework: a Test Target, which has a Test Bundle
- Test Bundles can have multiple test classes, which can be used to group tests (E.g., one class for each class being tested)
- Each test is a method, prefixed with 'test’
- Each method/test must prepare and cleanup any variables, structs, and any objects that it needs
  - If common to all tests, put in setup/teardown methods – but minimize this
- Asserts should be of the form: assert(test value, expected value, error message string, [parameters])
- Use the most descriptive/appropriate assert
  - E.g., XCTAssertEqual, XCTAssertNotEqual, XCTAssertEqualWithAccuracy, XCTAssertGreaterThan, XCTAssertNil, XCTAssertNotNil, XCTAssertTrue, XCTAssertFalse, etc.
- Can use XCTestExpectation's for testing async operations (async ‘flattening’ preferred instead, however)
- Can use Schemes to specify which tests to run in different scenarios
- “Enable Testability” build setting is enabled, and use “@testable import” in Unit Test classes