Testing Concurrent Applications

Testing is an important tool during the development process to create high quality applications. In the past, when concurrency was not such an important part of application architecture, testing was straightforward. Over the past few years it has become more and more important to use concurrent design patterns and we were challenged to develop new best practices to test them.

The main challenge of testing concurrent code is that the program or information flow is not reflected in the call stack any more. Functions do not return their result to the caller immediately, but deliver it later via callback functions, blocks, notifications, or similar mechanisms, which makes testing more difficult.

However, testing asynchronous code comes with the benefit of uncovering poor design decisions and facilitating clean implementations.

The Problem with Asynchronous Testing

Let’s first recall an example of a simple synchronous unit test. This method of a simple calculator should sum up two numbers:

+ (int)add:(int)a to:(int)b { return a + b; }

Testing this method is as simple as calling the method and comparing the result to the expected value. If the values don’t match, the test fails.

- (void)testAddition { int result = [Calculator add:2 to:2]; STAssertEquals(result, 4, nil); }

Now let’s change the method to return its result asynchronously via a completion block. We will also add a bug to the implementation, so that we can expect a failing test:

+ (int)add:(int)a to:(int)b block:(void(^)(int))block { [[NSOperationQueue mainQueue] addOperationWithBlock^{ block(a - b); // Buggy implementation }]; }

Of course this is a contrived example, but it reflects the general pattern you would use often if the operation would be more computationally intensive.

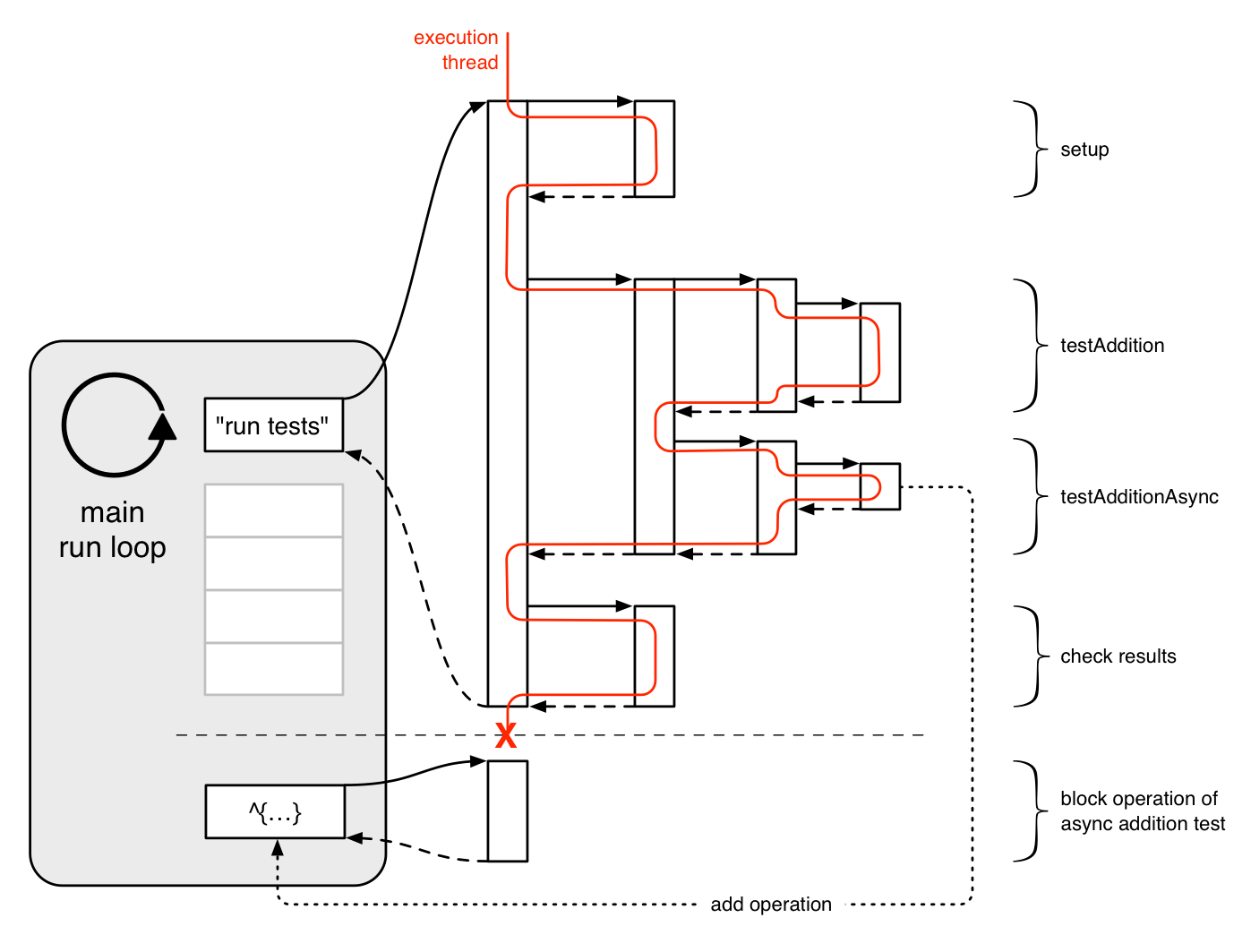
A naive approach to testing this method would just move the assertion into the completion block. However, such a test simply never fails, in spite of the bug in our implementation:

// don't use this code! - (void)testAdditionAsync { [Calculator add:2 to:2 block^(int result) { STAssertEquals(result, 4, nil); // Never fails! }]; }

Why doesn’t this assertion fail?

SenTestingKit Under the Hood

The testing framework used by Xcode 4 is based on [OCUnit](http://www.sente.ch/software/ocunit/), which allows us to have a closer look at the internals. To understand the problem with the asynchronous test, we need to have a look at the execution order of the different parts of the test suite. This diagram shows a simplified flow.



After the testing kit is started on the main run loop, it executes the following main steps:

1. It sets up a test suite containing all relevant tests (as specified e.g. in the project scheme).
2. It runs the suite, which internally invokes all methods of the test cases starting with*test*. This run returns an object, containing the results of each single test.
3. It exits the process by calling exit().

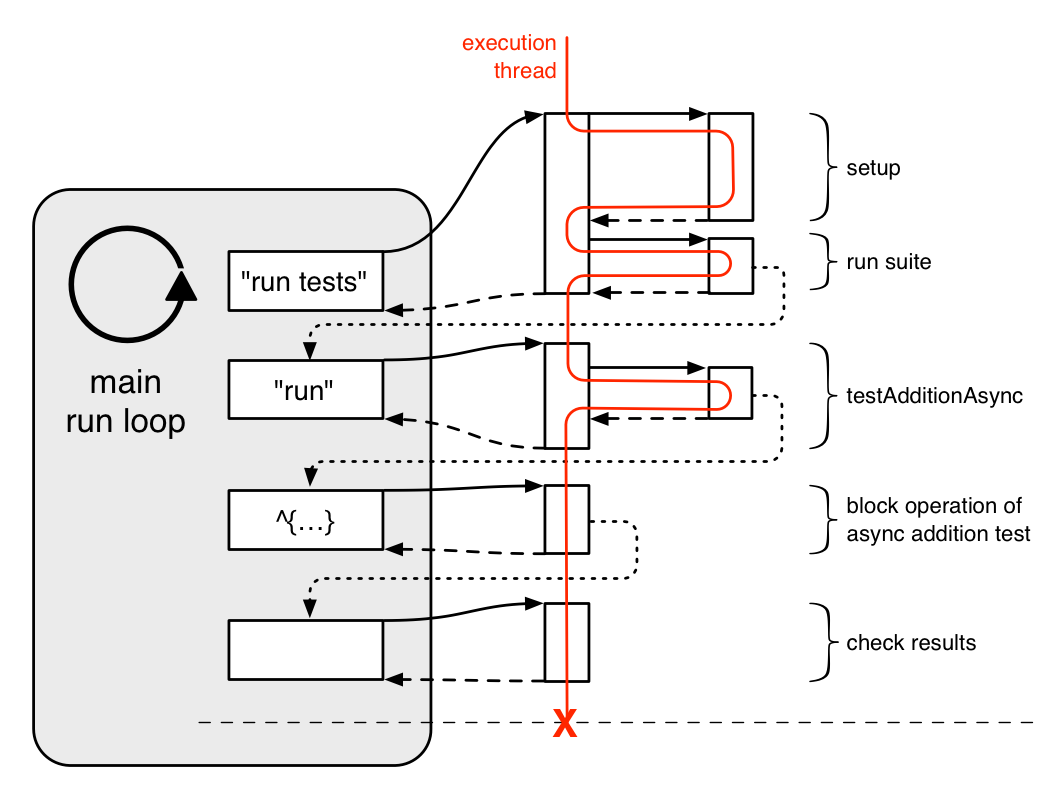
The interesting part is how each individual test is invoked. During the asynchronous test, the completion block containing the assertion gets enqueued on the main run loop. Since the testing framework exits the process after all tests run, this block never gets executed and therefore never causes the test to fail.

There are several approaches to solve this problem. But all of them have to run the main run loop and handle the enqueued operations before the test method returns and the framework checks the result.

[Kiwi](https://github.com/allending/Kiwi) uses a probe poller, which can be invoked within the test method. [GHUnit](https://github.com/gabriel/gh-unit/)provides a separate test class, which has to be prepared within the test method and which needs a notification at the end. In both cases we have to write some code, which ensures that the test method will not return until the test finishes.

Async Extension to SenTestingKit

Our solution to this problem is an [extension](https://github.com/nxtbgthng/SenTestingKitAsync) to the built-in testing kit, which winds up the synchronous execution on the stack and enqueues each part as a block on the main queue. As you can see in the diagram below, the block that reports the success or failure of the asynchronous test is enqueued before the results of the entire suite are checked. This execution order allows us to fire up a test and wait for its result.



To give the framework a hint that a test should be treated as asynchronous, the method name has to end with **Async**. Furthermore, in asynchronous tests, we have to report the success of the test case manually and include a timeout, in case the completion block never gets called. We can rewrite our faulty test from above like this:

- (void)testAdditionAsync { [Calculator add:2 to:2 block^(int result) { STAssertEquals(result, 4, nil); STSuccess(); // Calling this macro reports success }]; STFailAfter(2.0, @"Timeout"); }

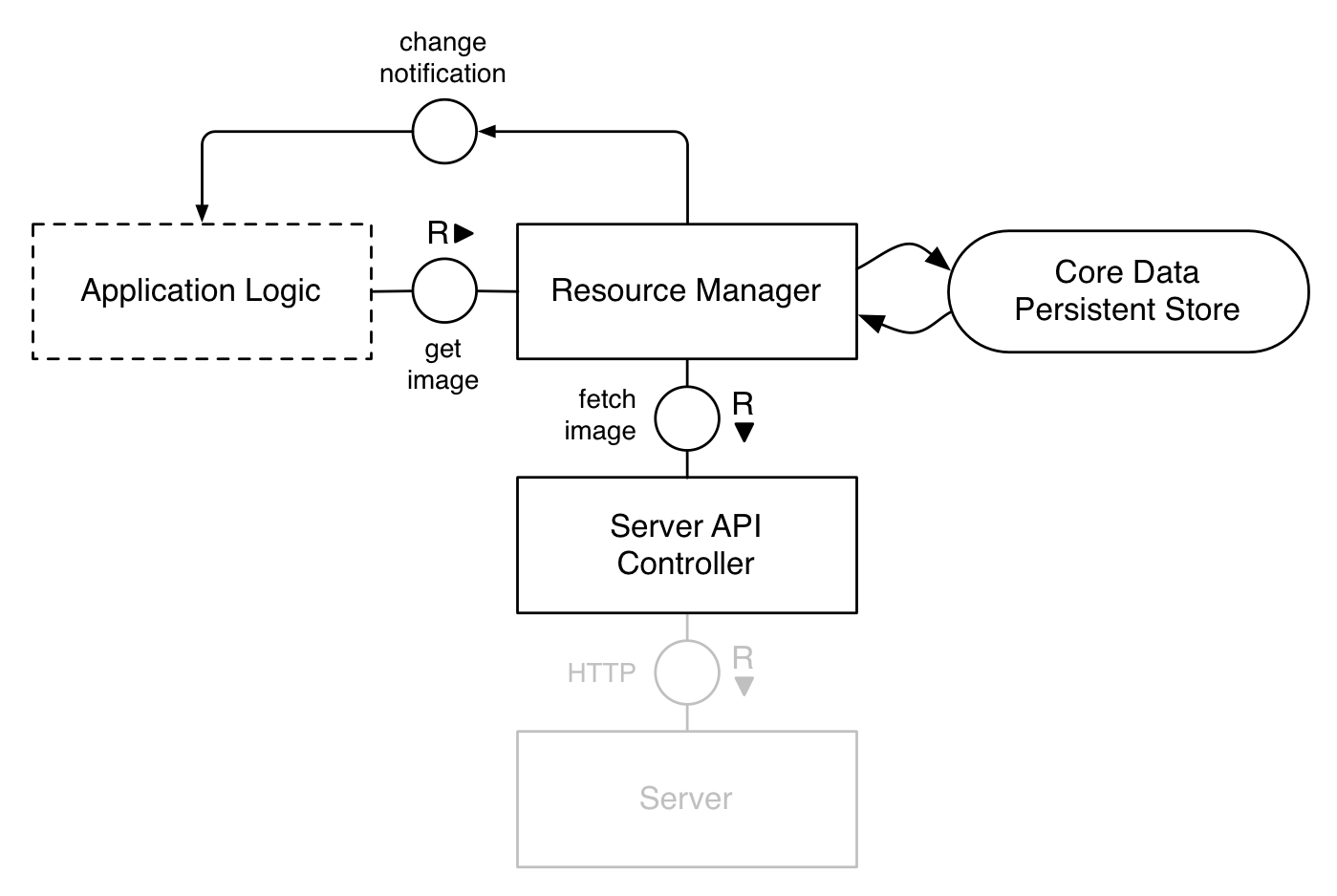
Designing Asynchronous Tests

As with their synchronous counterparts, asynchronous tests should always be a magnitude simpler than the implementation they are testing. Complex tests don’t promote better code quality, because the possibility of bugs in the tests increases. In a test-driven development process, simple tests let us think more clearly about the borders of components, their interfaces, and the expected behavior of the architecture.

Example Project

To put all this into practice, we create an example framework called [PinacotecaCore](https://github.com/objcio/issue-2-async-testing), which requests information of images from a hypothetical server. It has a resource manager, which is the developer-facing interface, providing a method to get an image object with an image id. In the background, the resource manager fetches the information from the server and updates the properties in the database.

Although this is only an example project for the sake of demonstration, it shares the pattern we use in several of our apps.



With this high level overview of the architecture we can dive into the tests of the framework. In general there are three components which should be tested:

1. the model layer
2. the server API controller, which abstracts the requests to the server
3. the resource manager, which manages the core data stack and ties the model layer and the API controller together

Model Layer

Tests should be synchronous whenever possible, and the model layer is a good example. As long as there are no complicated dependencies between different managed object contexts, the test cases should set up their own core data stack with a context on the main thread in which to execute their operations.

In this example, the [test case](https://github.com/objcio/issue-2-async-testing/blob/master/PinacotecaCore/PinacotecaCoreTests/PCModelLayerTests.m) sets up the core data stack in setUp, checks if the entity description for PCImage is present, creates an object with the constructor, and updates its values. As this has nothing to do with asynchronous testing, we won’t go into further details here.

Server API Controller

The second building block of the architecture is the server API controller. It contains the logic to manage the mapping of the server API to the model and handles the requests. In general, we want to evaluate the behavior of the following method:

- [PCServerAPIController fetchImageWithId:queue:completionHandler:]

It should be called with the id of an image and call the completion handler on the given queue.

Because the server doesn’t exist yet, and because it’s a good habit, we will stub the network request with [OHHTTPStubs](https://github.com/AliSoftware/OHHTTPStubs). With the newest version, the project can contain a bundle with example responses, which will be delivered to the client.

To stub a request, OHHTTPStubs has to be configured either in our test setup or in the test itself. First we have to load the bundle containing the responses:

NSURL \*url = [[NSBundle bundleForClass:[self class]] URLForResource:@"ServerAPIResponses" withExtension:@"bundle"]; NSBundle \*bundle = [NSBundle url];

Then we can load the response from the bundle and specify for which request it should be returned:

OHHTTPStubsResponse \*response; response = [OHHTTPStubsResponse responseNamed:@"images/123" fromBundle:responsesBundle responseTime:0.1]; [OHHTTPStubs stubRequestsPassingTest:^BOOL(NSURLRequest \*request) { return YES /\* true, if it's the expected request \*/; } withStubResponse:^OHHTTPStubsResponse \*(NSURLRequest \*request) { return response; }];

With this setup, the simplified version of the [API controller test](https://github.com/objcio/issue-2-async-testing/blob/master/PinacotecaCore/PinacotecaCoreTests/PCServerAPIControllerTests.m) looks like this:

- (void)testFetchImageAsync { [self.server fetchImageWithId:@"123" queue:[NSOperationQueue mainQueue] completionHandler:^(id imageData, NSError \*error) { STAssertEqualObjects([NSOperationQueue currentQueue], queue, nil); STAssertNil(error, [error localizedDescription]); STAssertTrue([imageData isKindOfClass:[NSDictionary class]], nil); // Check the values of the returned dictionary. STSuccess(); }]; STFailAfter(2.0, nil); }

Resource Manager

The last component is the resource manager, which ties the model layer and the API controller together and manages the core data stack. Here we want to test the method to get an image object:

-[PCResourceManager imageWithId:usingManagedObjectContext:queue:updateHandler:]

This method should return an image object for the given id. If this image is not in the database, it will return a new object containing only the id and call the API controller to request the detailed information.

Since the test of the resource manager should not depend on the API controller, we will stub it with [OCMock](http://ocmock.org/), which is ideal for partial stubs of methods. This is done in the [resource manager test](https://github.com/objcio/issue-2-async-testing/blob/master/PinacotecaCore/PinacotecaCoreTests/PCResourceManagerTests.m):

OCMockObject \*mo; mo = [OCMockObject partialMockForObject:self.resourceManager.server]; id exp = [[serverMock expect] andCall:@selector(fetchImageWithId:queue:completionHandler:) onObject:self]; [exp fetchImageWithId:OCMOCK\_ANY queue:OCMOCK\_ANY completionHandler:OCMOCK\_ANY];

Instead of calling the real method of the API controller, the test will use the method implemented in the test case itself.

With this in place, the test of the resource manager is straightforward. It calls the manager to get a resource, which internally will call the stubbed method on the API controller. There we can check if the controller is called with the correct parameters. After invoking the result handler, the resource manager updates the model and will call the result handler of our test.

- (void)testGetImageAsync { NSManagedObjectContext \*ctx = self.resourceManager.mainManagedObjectContext; \_\_block PCImage \*img; img = [self.resourceManager imageWithId:@"123" usingManagedObjectContext:ctx queue:[NSOperationQueue mainQueue] updateHandler:^(NSError \*error) { // Check if the error is nil and // if the image has been updated. STSuccess(); }]; STAssertNotNil(img, nil); STFailAfter(2.0, @"Timeout"); }

Conclusion

Testing applications using concurrent design patterns can be challenging at first, but once you understand the differences and establish best practices, it is easy and a lot of fun.

At [nxtbgthng](http://nxtbgthng.com/) we are using the process described, [SenTestingKitAsync](https://github.com/nxtbgthng/SenTestingKitAsync), on a daily basis. But the other approaches, like [Kiwi](https://github.com/allending/Kiwi) or [GHUnit](https://github.com/gabriel/gh-unit/), are also good ways of doing asynchronous testing. You should try them all, find your preferred tool, and start testing asynchronously.