Test Driven Development

*Sourced from Test Driven iOS Development by Graham Lee

Test Driven Development

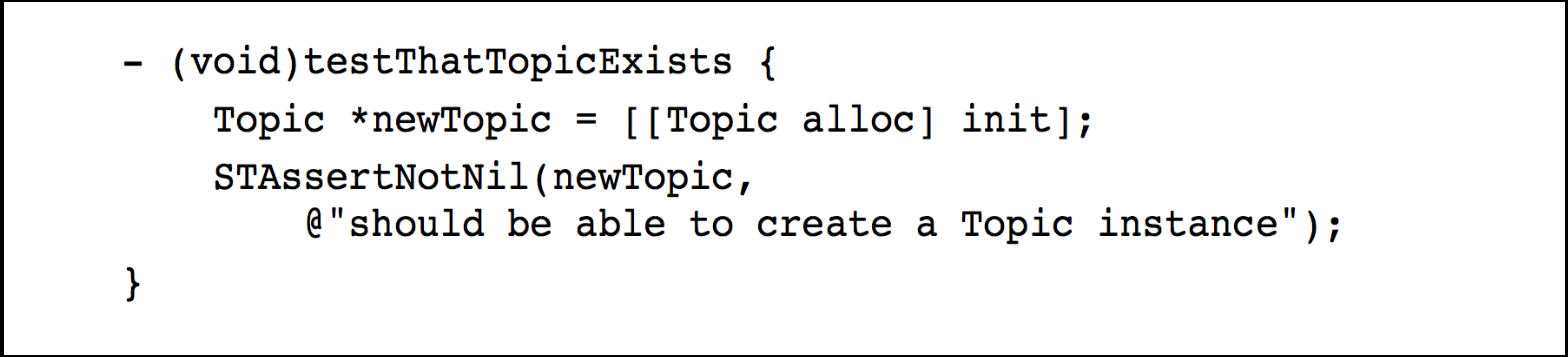
- **Test First** : Developers are encouraged to write tests before writing code that will be tested.
- **Red, Green, Refactor**: Writing a failing test that encapsulates the desired behavior of the code you have not yet written
- **App Design**: Getting an idea of what the features of the app are how they all fit together, also known as System Metaphor.
- **Independent Tests**: Each class should have an associated unit test class that tests methods of that class. Being unit tests, we want to minimize the dependencies of the class we are testing . To help with this, we can use **mock objects**.

TDD Example

- The author of “Test-Driven iOS Development” uses a stack overflow client, somewhat similar to ours, to teach how TDD can work with an iOS app.
- Lets take a look at some of his examples

TDD for your Model

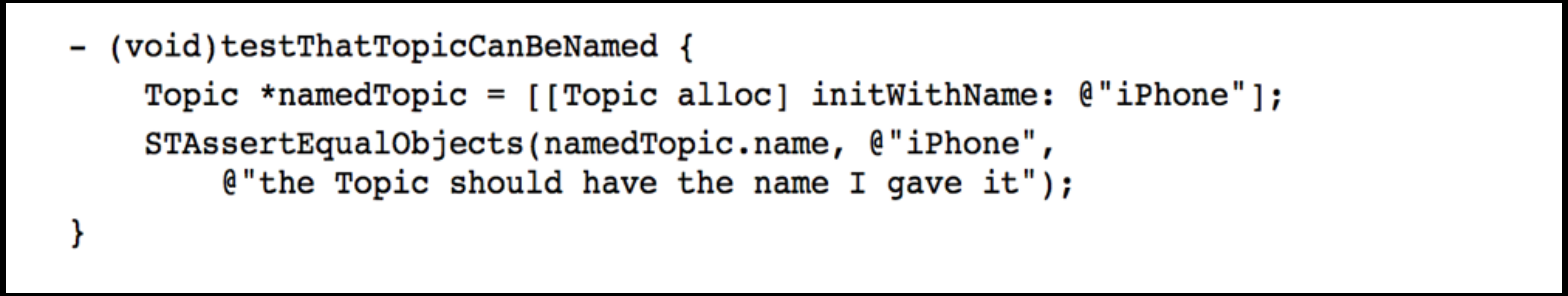
- He starts off with a Topic model class
- Before even creating the class, he creates a test class for it and writes a single test that wont even compile:



```
- (void)testThatTopicExists {
    Topic *newTopic = [[Topic alloc] init];
    STAssertNotNil(newTopic,
        @"should be able to create a Topic instance");
}
```

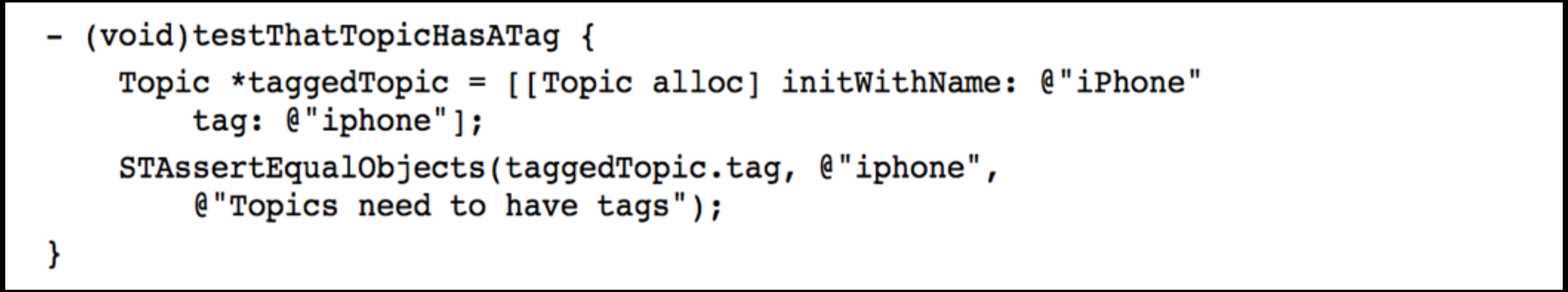
TDD for your Model

- He then creates the class and imports Topic.h into the test class and the test now passes!
- He then writes a test to test an initializer that takes in a string to be used as the topics name:



```
- (void)testThatTopicCanBeNamed {
    Topic *namedTopic = [[Topic alloc] initWithName: @"iPhone"];
    STAssertEqualObjects(namedTopic.name, @"iPhone",
        @"the Topic should have the name I gave it");
}
```

- He then adds the name property to the Topic class and also adds the custom init as well.
- Finally he writes a test to check if a tag can be set as well:



```
- (void)testThatTopicHasATag {
    Topic *taggedTopic = [[Topic alloc] initWithName: @"iPhone"
        tag: @"iphone"];
    STAssertEqualObjects(taggedTopic.tag, @"iphone",
        @"Topics need to have tags");
}
```

- And then writes the code to make that test pass

Red,Green,Refactor

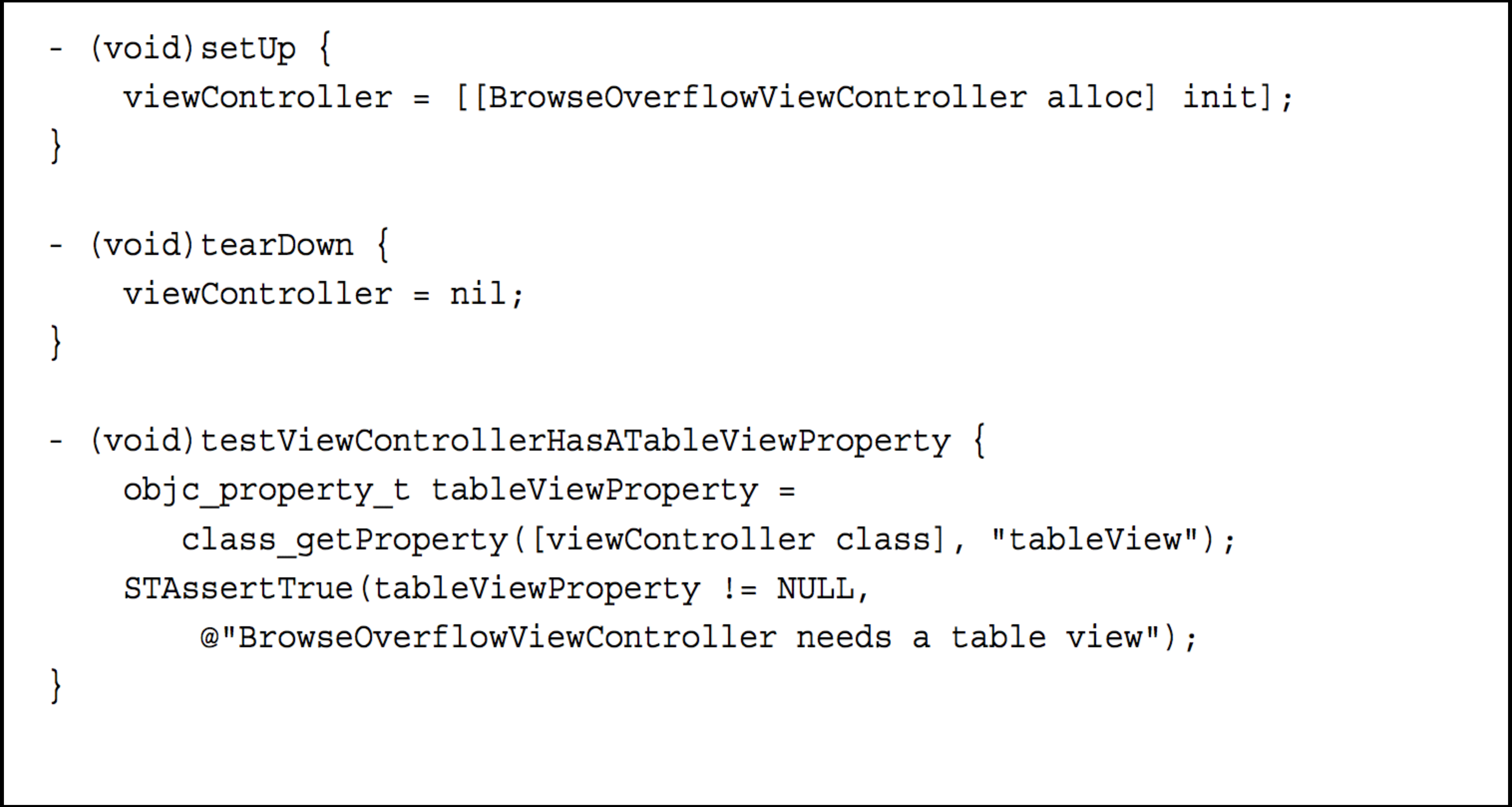
- After taking those test from red to green, now is a great time to see if any refactoring can be done.
- The author concludes that he will never use the init with only the name parameter, so he deletes and refactors his test to only use the initWithName:andTag:
- He also creates a topic property that all the tests can use.

```
- (void)setUp {
    topic = [[Topic alloc] initWithName: @"iPhone" tag: @"iphone"];
}

- (void)tearDown {
    topic = nil;
}

- (void)testThatTopicExists {
    STAssertNotNil(topic, @"should be able to create a Topic instance");
}

- (void)testThatTopicCanBeNamed {
    STAssertEqualObjects(topic.name, @"iPhone",
        @"the Topic should have the name I gave it");
}

- (void)testThatTopicHasATag {
    STAssertEqualObjects(topic.tag, @"iphone",
        @"the Topic should have the tag I gave it");
}
```

Testing the Controller layer

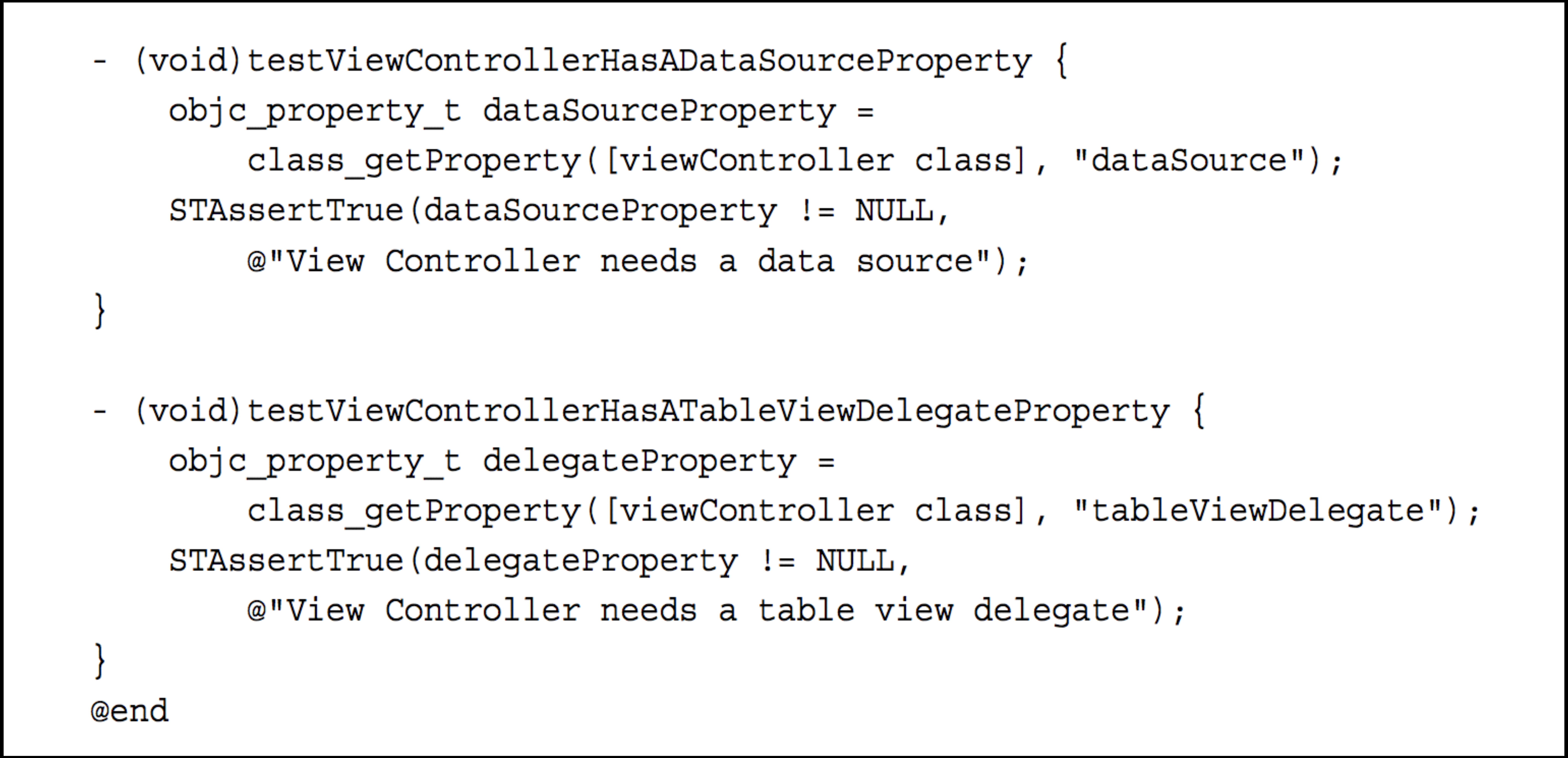
- The author then concludes he really only needs one view controller for this app, since everything is displayed in a table view.
- He creates a class called BrowseOverflowViewController, which is going to have tableView, a dataSource, and a tableViewDelegate properties
- He then uses some objective-c runtime magic to write a few initial tests

Testing the Controller layer



```
- (void)setUp {
    viewController = [[BrowseOverflowViewController alloc] init];
}

- (void)tearDown {
    viewController = nil;
}

- (void)testViewControllerHasATableViewProperty {
    objc_property_t tableViewProperty =
        class_getProperty([viewController class], "tableView");
    STAssertTrue(tableViewProperty != NULL,
         @"BrowseOverflowViewController needs a table view");
}
```

Testing the Controller layer

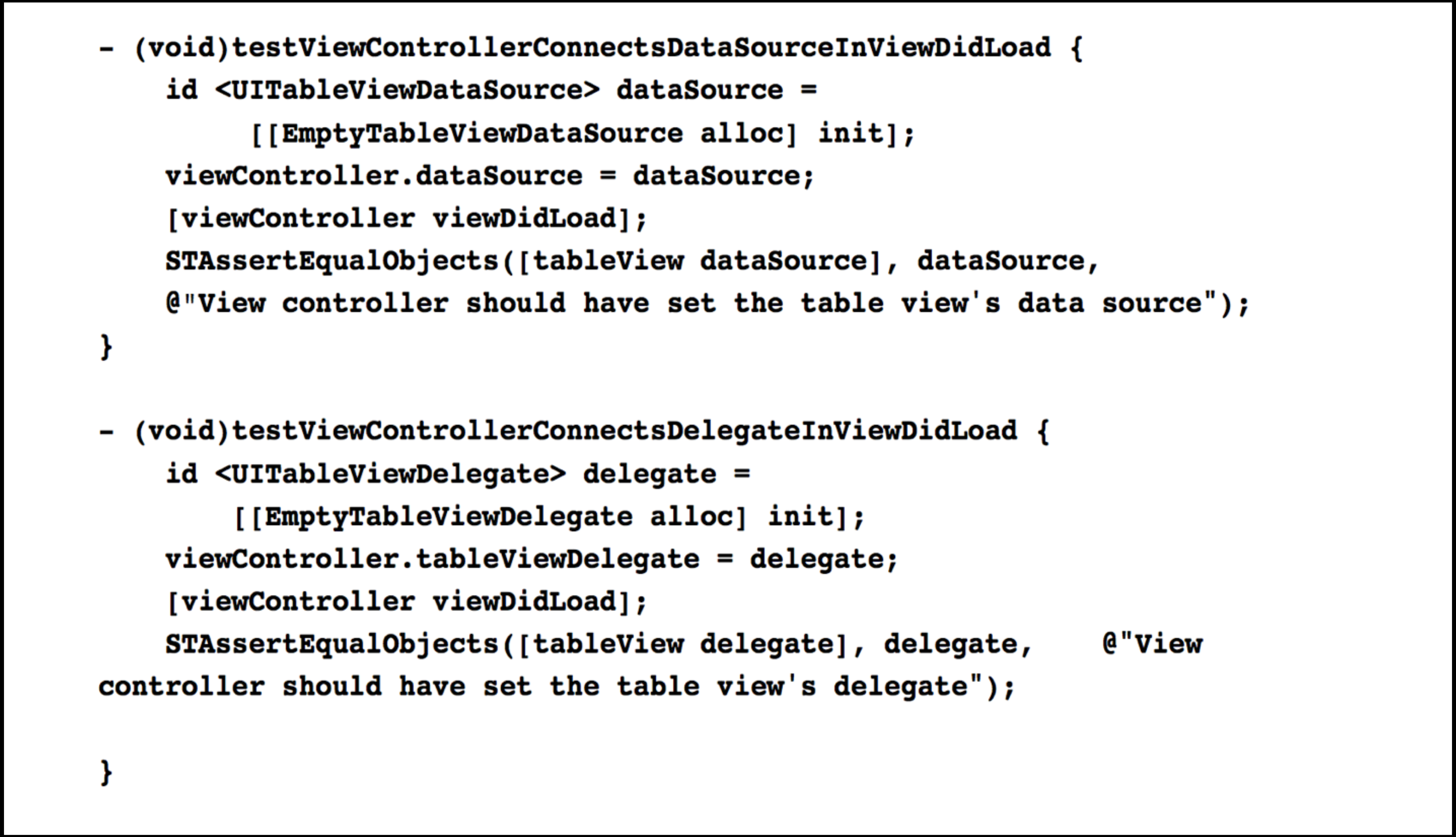


```
- (void)testViewControllerHasADataSourceProperty {
    objc_property_t dataSourceProperty =
        class_getProperty([viewController class], "dataSource");
    STAssertTrue(dataSourceProperty != NULL,
        @"View Controller needs a data source");
}

- (void)testViewControllerHasATableViewDelegateProperty {
    objc_property_t delegateProperty =
        class_getProperty([viewController class], "tableViewDelegate");
    STAssertTrue(delegateProperty != NULL,
        @"View Controller needs a table view delegate");
}
@end
```

Testing the Controller layer

- So the main responsibility of the view controller is to connect the table view with the proper data sources and delegates.
- So he next writes tests for that initial behavior

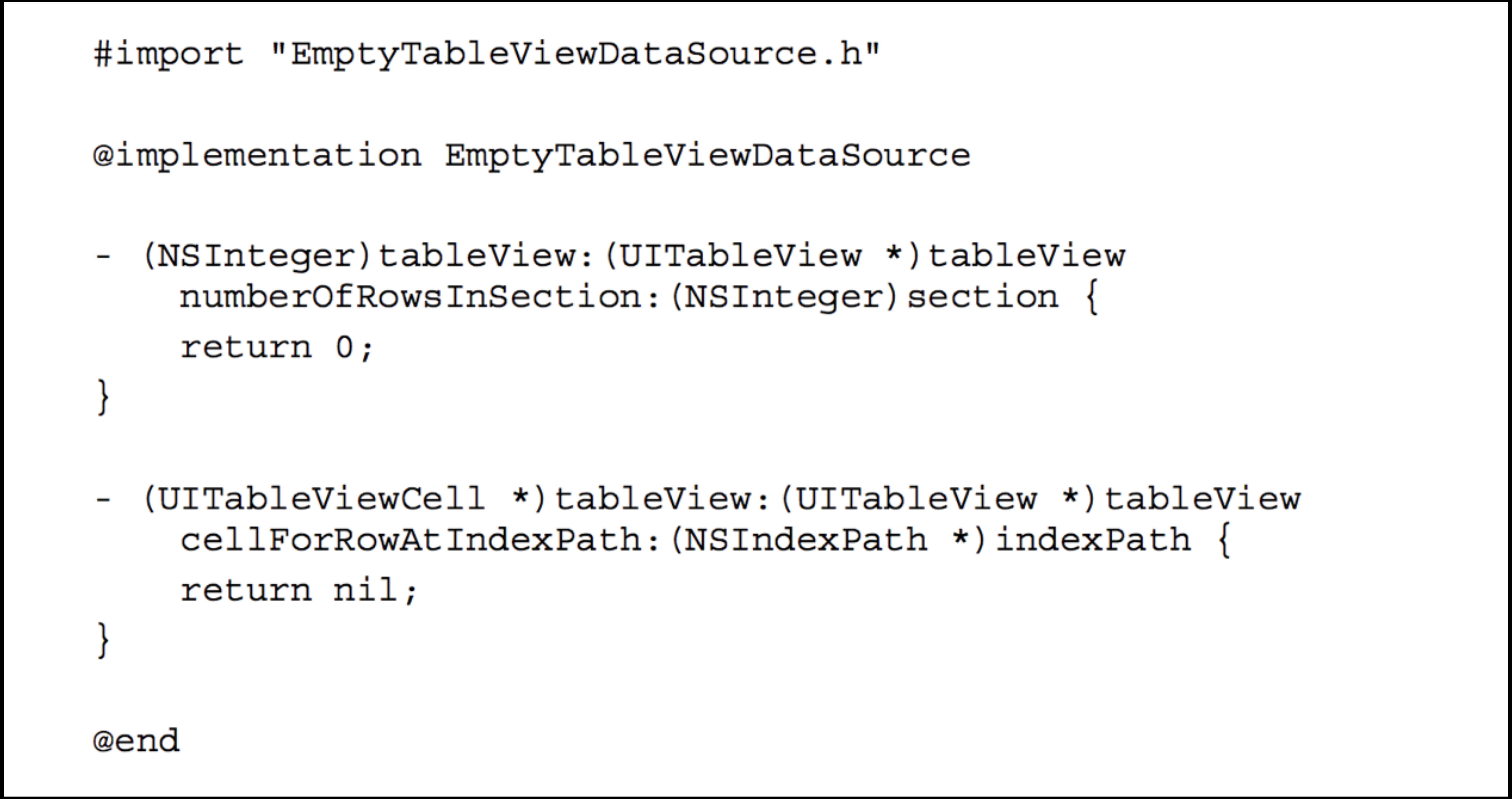
Testing the Controller layer



```
- (void)testViewControllerConnectsDataSourceInViewDidLoad {
    id <UITableViewDataSource> dataSource =
         [[EmptyTableViewDataSource alloc] init];
    viewController.dataSource = dataSource;
    [viewController viewDidLoad];
    STAssertEqualObjects([tableView dataSource], dataSource,
    @"View controller should have set the table view's data source");
}

- (void)testViewControllerConnectsDelegateInViewDidLoad {
    id <UITableViewDelegate> delegate =
        [[EmptyTableViewDelegate alloc] init];
    viewController.tableViewDelegate = delegate;
    [viewController viewDidLoad];
    STAssertEqualObjects([tableView delegate], delegate,    @"View
controller should have set the table view's delegate");

}
```

Testing the Controller layer

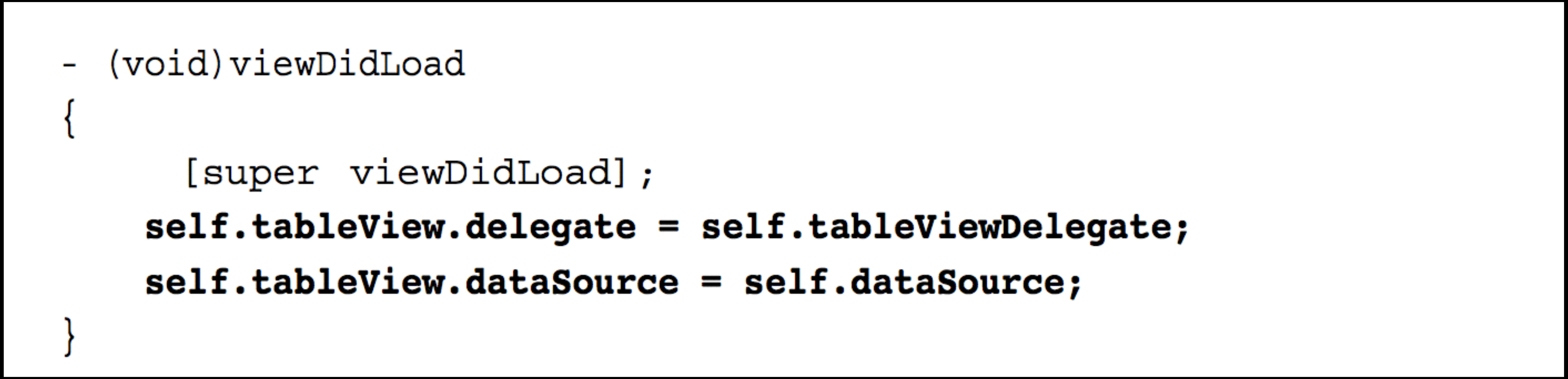
- The initial implementation of EmptyTableViewDataSource is very minimal:



```
#import "EmptyTableViewDataSource.h"

@implementation EmptyTableViewDataSource

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    return nil;
}

@end
```

Testing the Controller layer

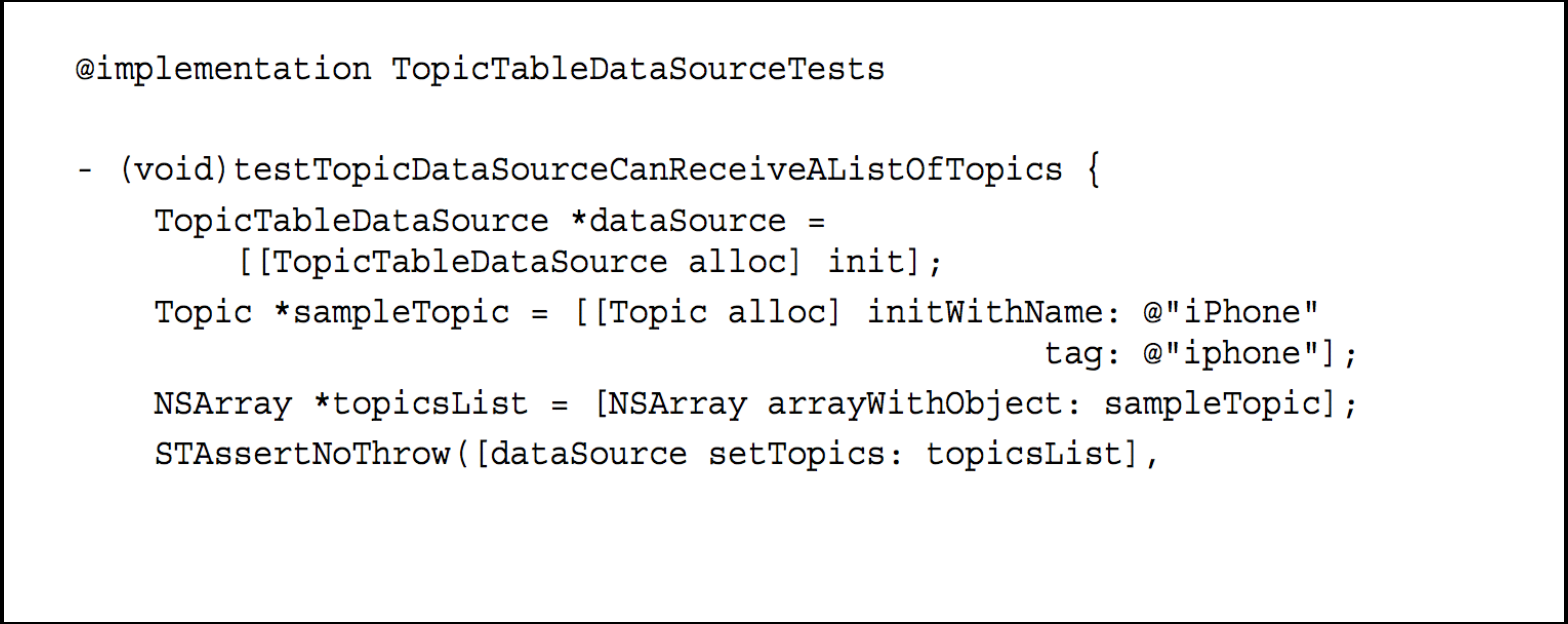
- Finally, to make these tests pass, he simply adds the two lines of code we have seen so many times!



```
- (void)viewDidLoad
{
      [super viewDidLoad];
    self.tableView.delegate = self.tableViewDelegate;
    self.tableView.dataSource = self.dataSource;
}
```

Testing the Controller layer

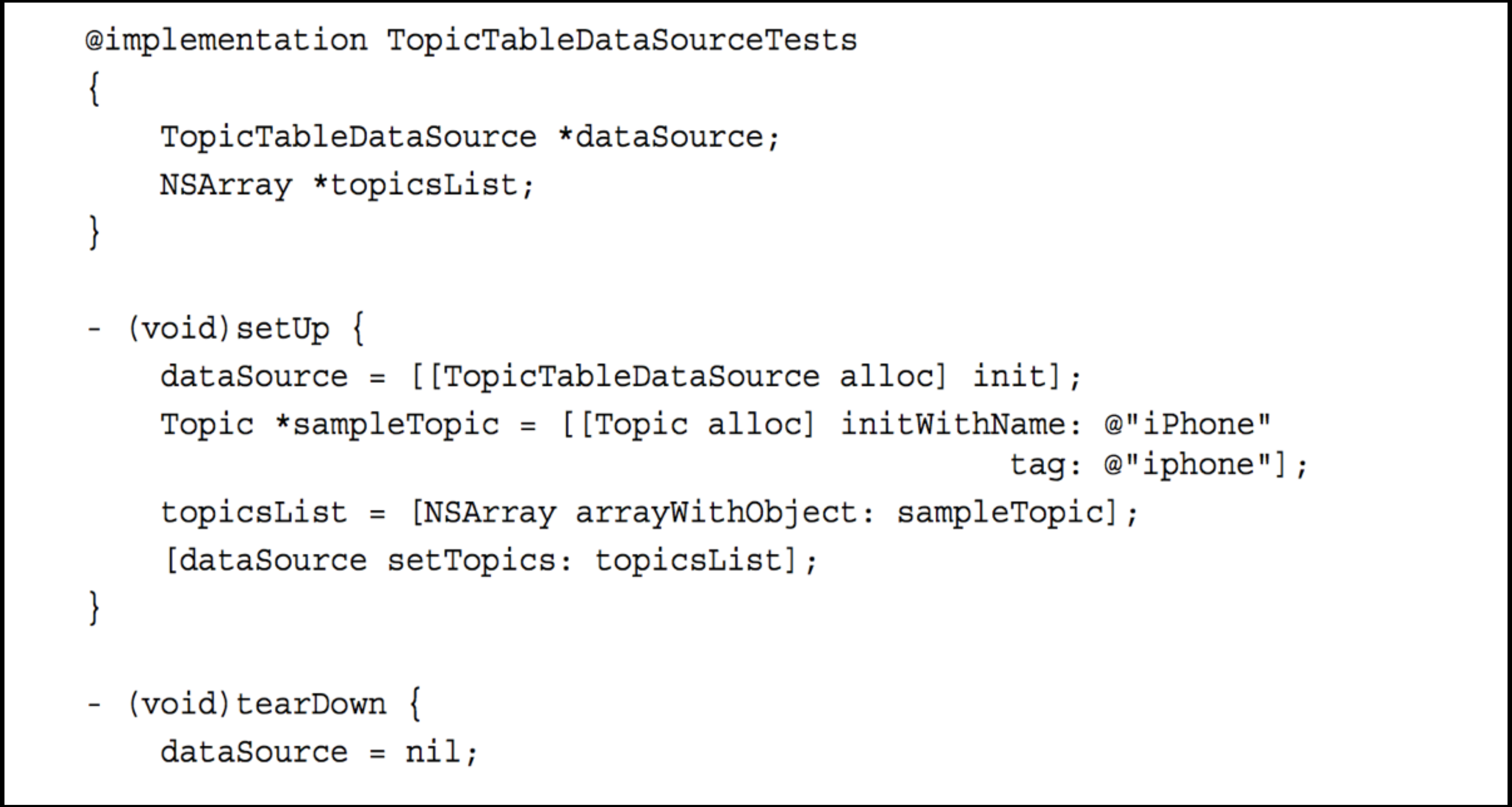
- He then creates a real TopicTableDataSource class and writes a test for it to store an array of topics



```
@implementation TopicTableDataSourceTests

- (void)testTopicDataSourceCanReceiveAListOfTopics {
    TopicTableDataSource *dataSource =
        [[TopicTableDataSource alloc] init];
    Topic *sampleTopic = [[Topic alloc] initWithName: @"iPhone"
                                                 tag: @"iphone"];
    NSArray *topicsList = [NSArray arrayWithObject: sampleTopic];
    STAssertNoThrow([dataSource setTopics: topicsList],
```

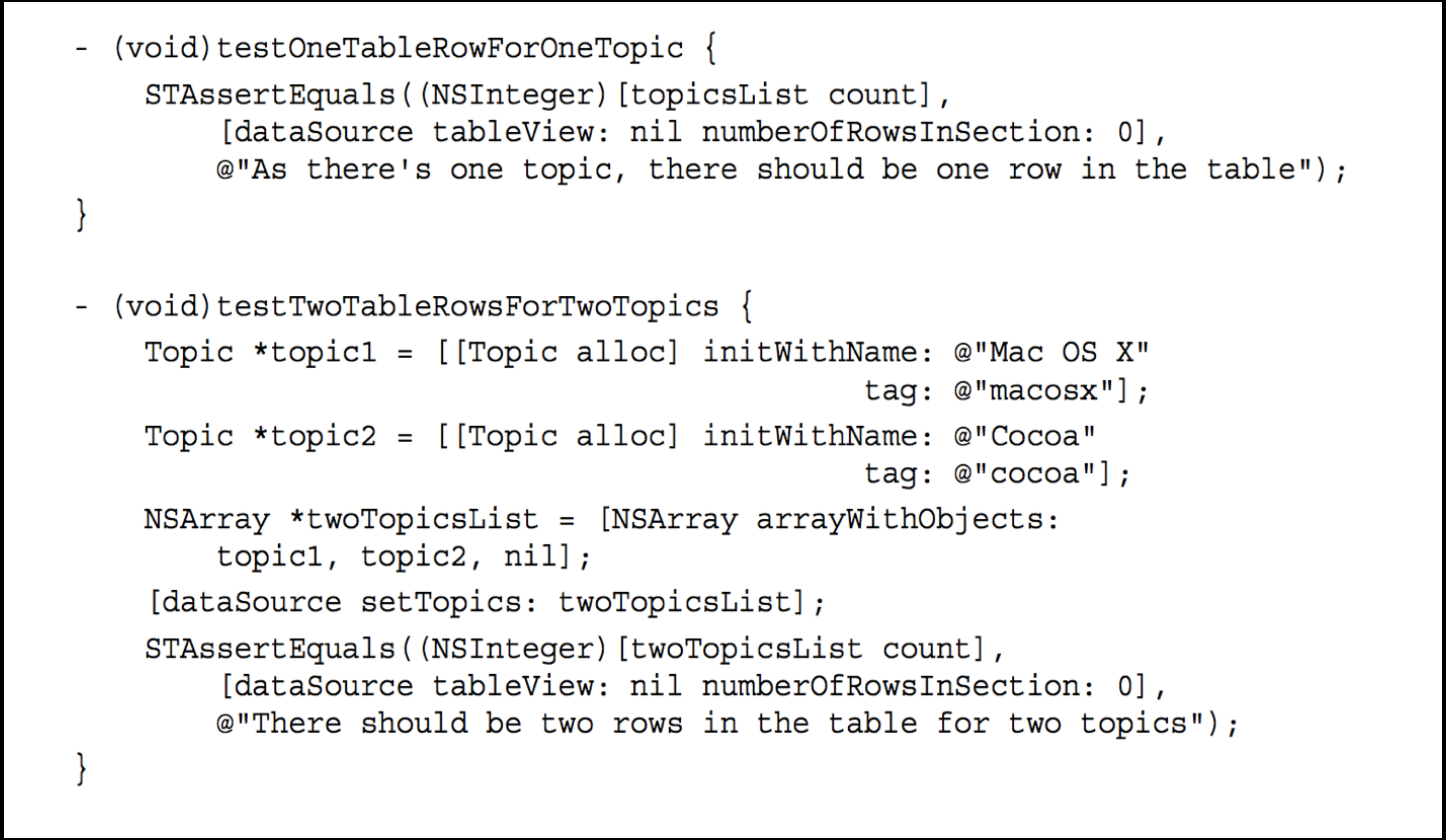
Testing the Controller layer

- He then refactors his tests



```
@implementation TopicTableDataSourceTests
{
    TopicTableDataSource *dataSource;
    NSArray *topicsList;
}

- (void)setUp {
    dataSource = [[TopicTableDataSource alloc] init];
    Topic *sampleTopic = [[Topic alloc] initWithName: @"iPhone"
                                                 tag: @"iphone"];
    topicsList = [NSArray arrayWithObject: sampleTopic];
    [dataSource setTopics: topicsList];
}

- (void)tearDown {
    dataSource = nil;
```

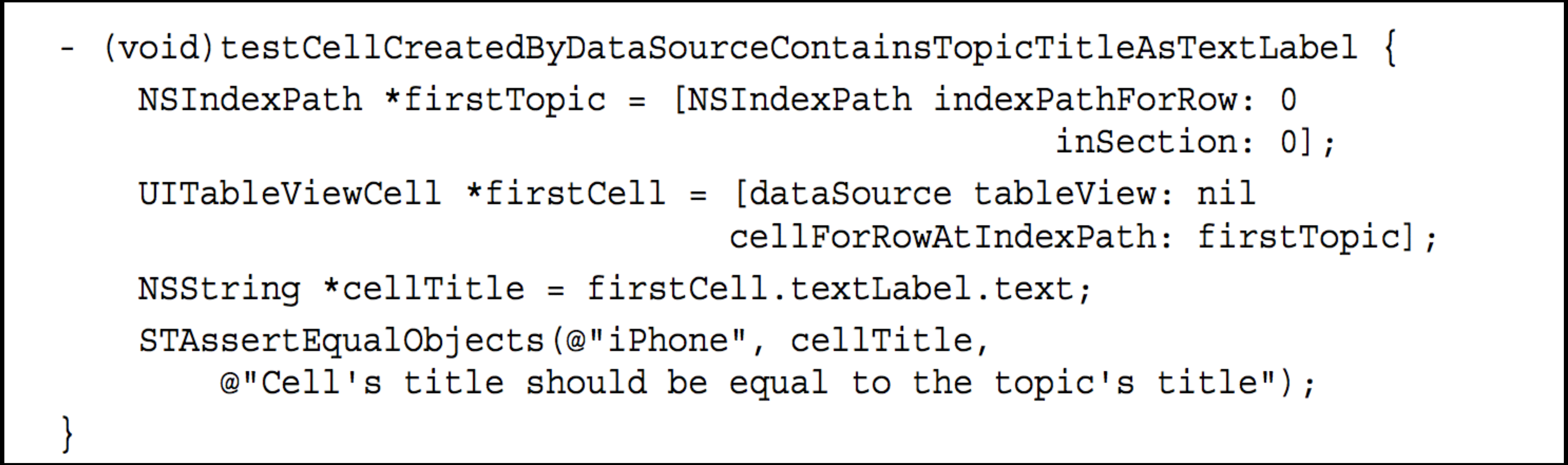
Testing the Controller layer



```
- (void)testOneTableRowForOneTopic {
    STAssertEquals((NSInteger)[topicsList count],
        [dataSource tableView: nil numberOfRowsInSection: 0],
        @"As there's one topic, there should be one row in the table");
}

- (void)testTwoTableRowsForTwoTopics {
    Topic *topic1 = [[Topic alloc] initWithName: @"Mac OS X"
                                            tag: @"macosx"];
    Topic *topic2 = [[Topic alloc] initWithName: @"Cocoa"
                                            tag: @"cocoa"];
    NSArray *twoTopicsList = [NSArray arrayWithObjects:
        topic1, topic2, nil];
    [dataSource setTopics: twoTopicsList];
    STAssertEquals((NSInteger)[twoTopicsList count],
        [dataSource tableView: nil numberOfRowsInSection: 0],
        @"There should be two rows in the table for two topics");
}
```

Testing the Controller layer

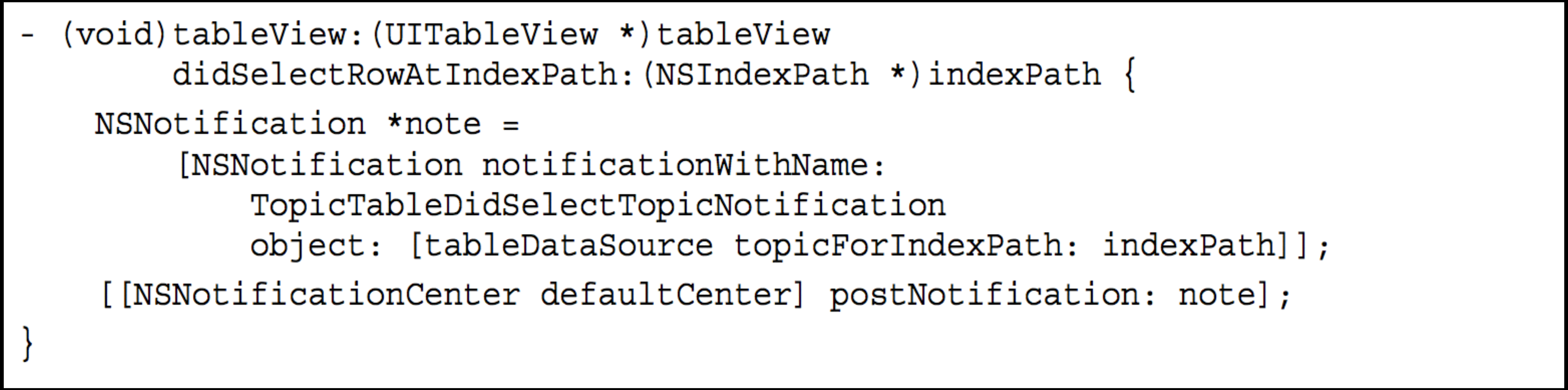
- Heres how he tests cell creation by the datasource



```
- (void)testCellCreatedByDataSourceContainsTopicTitleAsTextLabel {
    NSIndexPath *firstTopic = [NSIndexPath indexPathForRow: 0
                                                 inSection: 0];
    UITableViewCell *firstCell = [dataSource tableView: nil
                             cellForRowAtIndexPath: firstTopic];
    NSString *cellTitle = firstCell.textLabel.text;
    STAssertEqualObjects(@"iPhone", cellTitle,
        @"Cell's title should be equal to the topic's title");
}
```

Testing the Controller layer

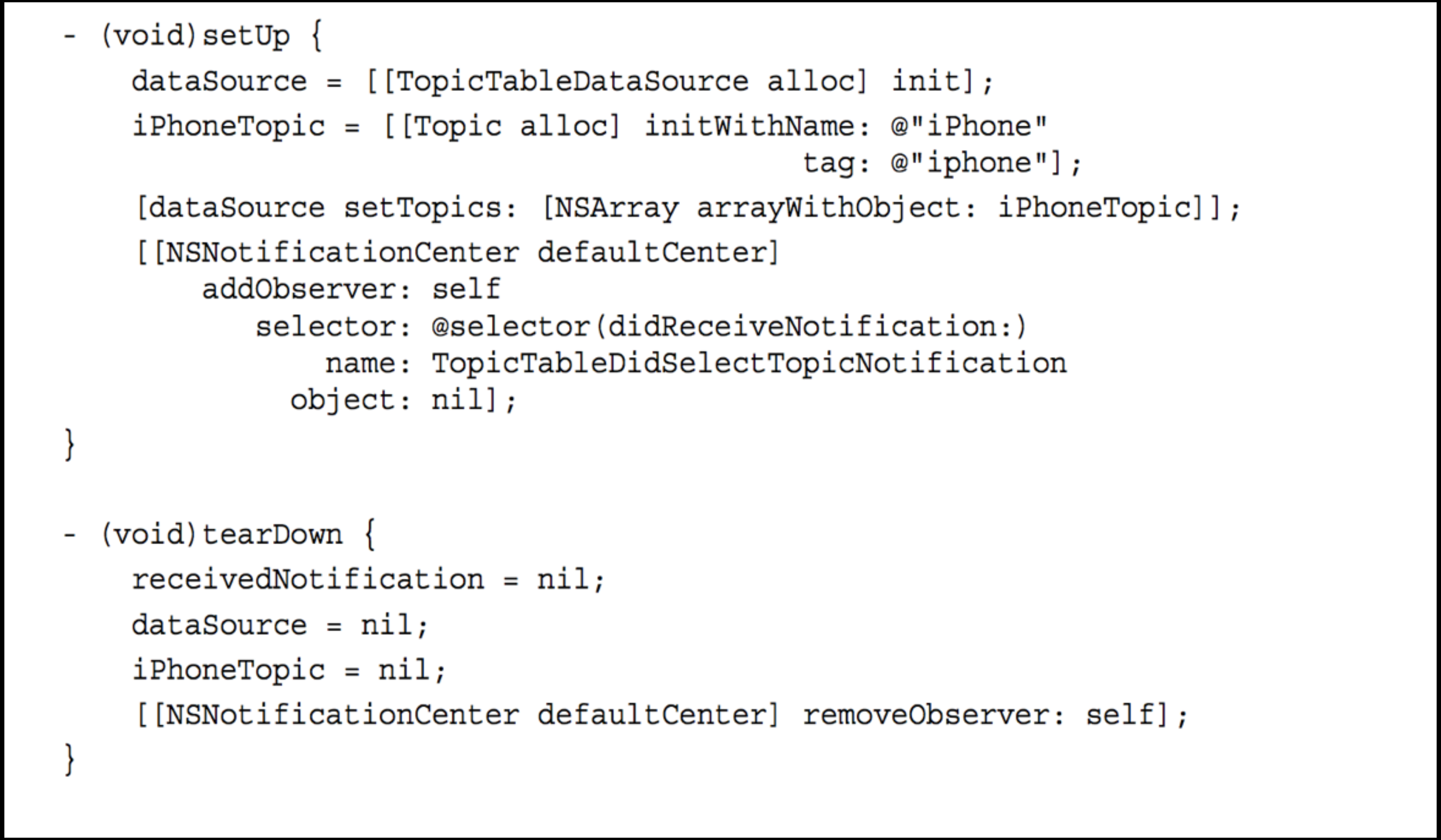
- He then uses notification center to notify the view controller when the user selects a cell



```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSNotification *note =
        [NSNotification notificationWithName:
            TopicTableDidSelectTopicNotification
            object: [tableDataSource topicForIndexPath: indexPath]];
    [[NSNotificationCenter defaultCenter] postNotification: note];
}
```

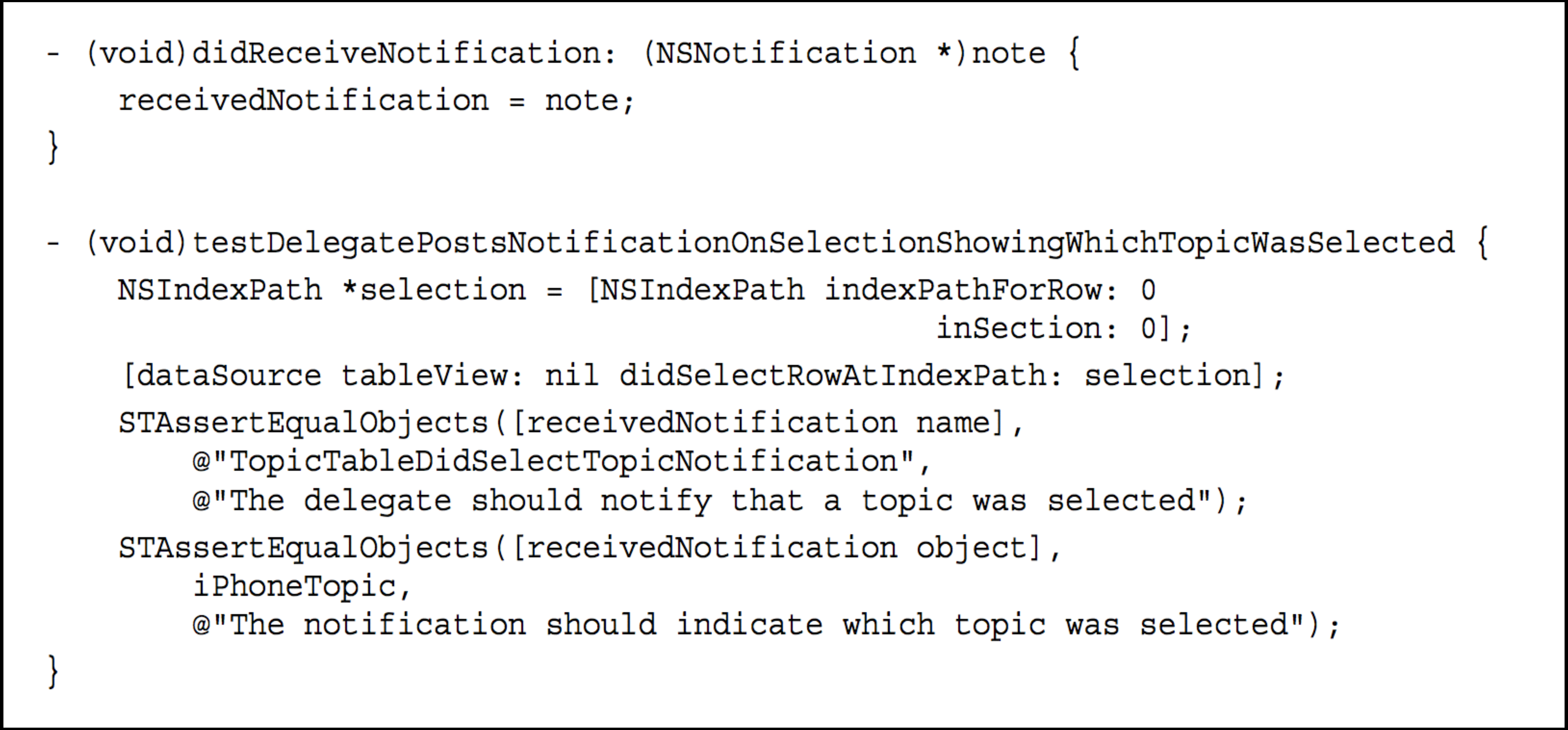
Testing the Controller layer

- And here is the test for that



```
- (void)setUp {
    dataSource = [[TopicTableDataSource alloc] init];
    iPhoneTopic = [[Topic alloc] initWithName: @"iPhone"
                                          tag: @"iphone"];
    [dataSource setTopics: [NSArray arrayWithObject: iPhoneTopic]];
    [[NSNotificationCenter defaultCenter]
        addObserver: self
           selector: @selector(didReceiveNotification:)
               name: TopicTableDidSelectTopicNotification
             object: nil];
}

- (void)tearDown {
    receivedNotification = nil;
    dataSource = nil;
    iPhoneTopic = nil;
    [[NSNotificationCenter defaultCenter] removeObserver: self];
}
```

Testing the Controller layer



```
- (void)didReceiveNotification: (NSNotification *)note {
    receivedNotification = note;
}

- (void)testDelegatePostsNotificationOnSelectionShowingWhichTopicWasSelected {
    NSIndexPath *selection = [NSIndexPath indexPathForRow: 0
                                                inSection: 0];
    [dataSource tableView: nil didSelectRowAtIndexPath: selection];
    STAssertEqualObjects([receivedNotification name],
        @"TopicTableDidSelectTopicNotification",
        @"The delegate should notify that a topic was selected");
    STAssertEqualObjects([receivedNotification object],
        iPhoneTopic,
        @"The notification should indicate which topic was selected");
}
```

OCMock

- OCMock is a framework used to create mock objects for objective-c
- OCMock is great for creating stub objects, aka objects that are setup to return pre-determined values for specific method invokations.

OCMock example

- Lets look at an example of using OCMock on a twitter application similar to our week 1 app.
- In this example we have 3 primary classes:
  - Controller
  - TwitterConnection (handles calls to the API)
  - TweetView (displays a tweet)

OCMock example

```
@interface Controller

@property(retain) TwitterConnection *connection;
@property(retain) TweetView *tweetView;

- (void)updateTweetView;

@end
```

Controller has references
to the connection
and the tweet view

```
@interface TwitterConnection

- (NSArray *)fetchTweets;

@end
```

TwitterConnection has a
method that returns tweets

```
@interface TweetView

- (void)addTweet:(Tweet *)aTweet;

@end
```

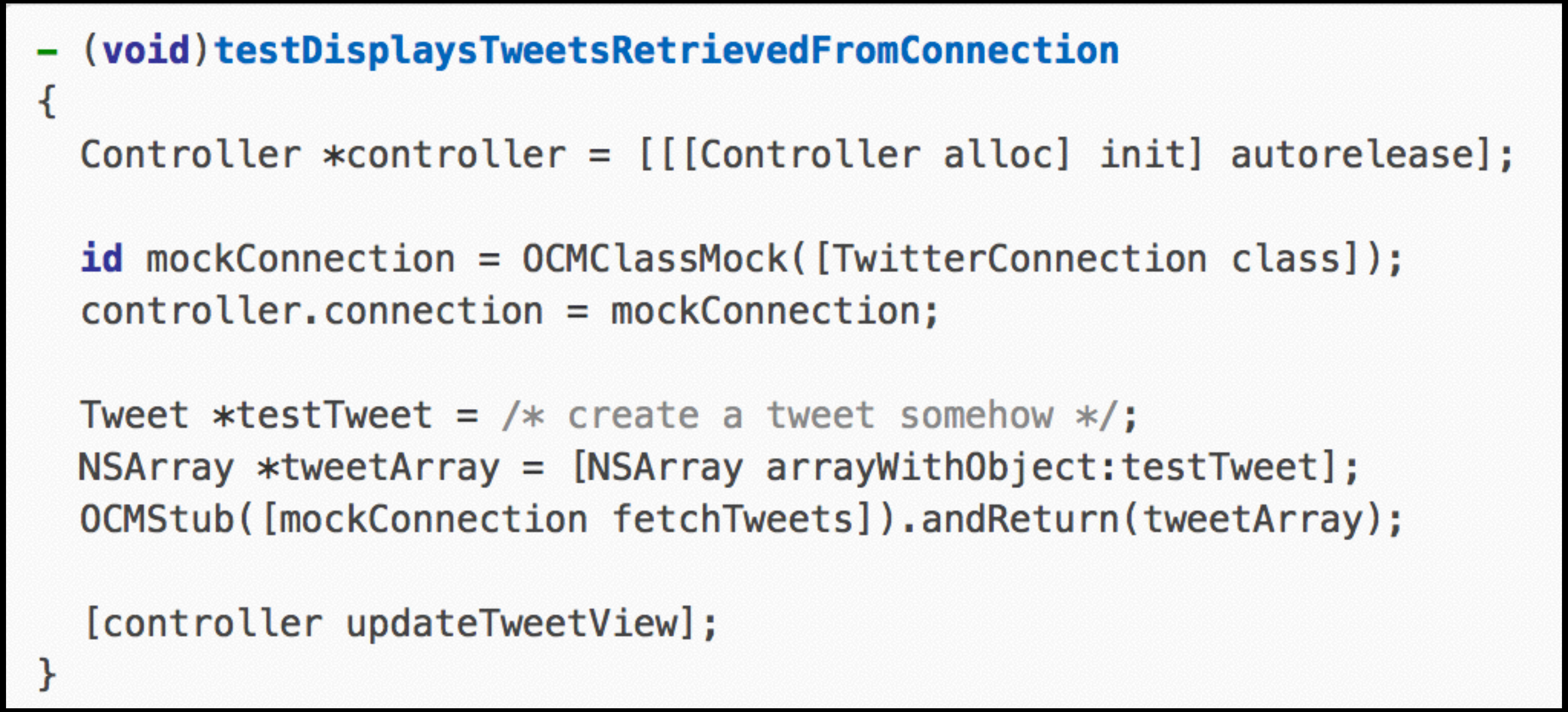
TweetView has a method
to add tweets

OCMock example

- When we want to write a test for the updateTweetView: method on the controller, we have to figure out what to do about the controller's dependancies on the twitter connection and the view.
- We could instantiate and use a real twitter connection object, but that introduces problems:
    - That would make the test slow because internet
    - Twitters server might have an error, but thats not what we are testing
    - Asynchronous testing can be tricky
- The solution is to create a stub of the the connection class!

Create a stub

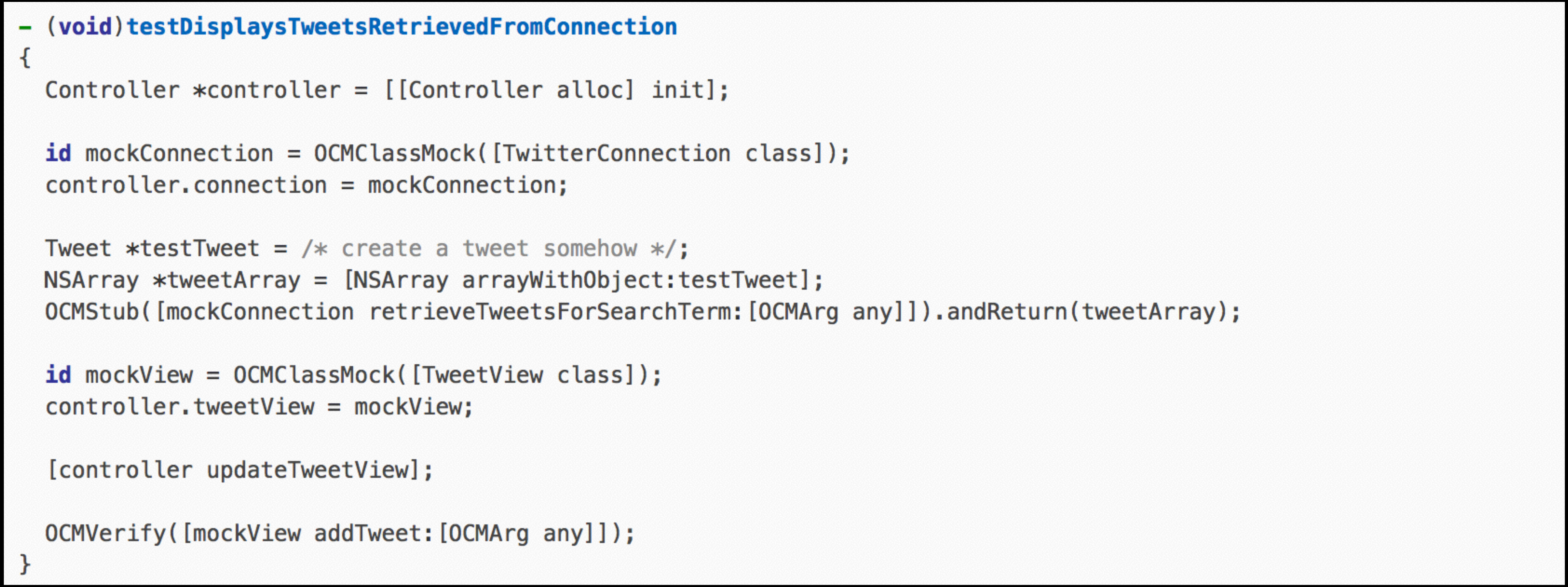
- Here what creating a stub of the twitter connection class, and giving it a predefined array to return when the method fetchTweets is called:



```
- (void)testDisplaysTweetsRetrievedFromConnection
{
  Controller *controller = [[[Controller alloc] init] autorelease];

  id mockConnection = OCMClassMock([TwitterConnection class]);
  controller.connection = mockConnection;

  Tweet *testTweet = /* create a tweet somehow */;
  NSArray *tweetArray = [NSArray arrayWithObject:testTweet];
  OCMStub([mockConnection fetchTweets]).andReturn(tweetArray);

  [controller updateTweetView];
}
```

Creating a Mock

- In contrast to stubs, which basically just return canned results, mocks are used to verify interactions.
- In our case we will want to verify that the controller makes the right calls to the tweet view.



```
- (void)testDisplaysTweetsRetrievedFromConnection
{
  Controller *controller = [[Controller alloc] init];

  id mockConnection = OCMClassMock([TwitterConnection class]);
  controller.connection = mockConnection;

  Tweet *testTweet = /* create a tweet somehow */;
  NSArray *tweetArray = [NSArray arrayWithObject:testTweet];
  OCMStub([mockConnection retrieveTweetsForSearchTerm:[OCMArg any]]).andReturn(tweetArray);

  id mockView = OCMClassMock([TweetView class]);
  controller.tweetView = mockView;

  [controller updateTweetView];

  OCMVerify([mockView addTweet:[OCMArg any]]);
}
```

Swift and Mocks

- OCMock doesn't support Swift at the moment
- OCMock relies on the exposed Obejctive-C Runtime to function, Swift’s runtime isn't exposed.
- But Swift is a bit more friendly with creating mock objects natively, since you can actually declare classes inside methods.
- So you can declare and instant ‘fake’ or mock classes inside your test methods