Unit Testing with OCUnit

Lecture 8
Overview

• Assuming Xcode 3

• Mac OS X Applications & iOS Applications
  • Logic tests (build-time)
  • Application tests (run-time)

• Automation
  • Building before testing, testing before running.
  • Input (e.g., simulating touches)
Why Test?

- Healthier Software
  - Confident to update code
- Design Improvements
  - Self-checking the usability of your API
- Solid Foundation
  - Building on stable/tested code
- Executable Documentation
  - Demonstrating usage of your API
- Accelerated Development Pace
  - Generates more quality code overall
OS X vs. iOS

• Both can do logic & application testing
• On Xcode 3, application testing within iOS is device only
• On Xcode 4 can be done in simulator
• (Almost) identical functionality/API
Mac OS X App. Testing

• Typically start by testing the model
• UI is testable, but more difficult challenging.
iOS: Logic & App. tests

- Logic Tests
  - Testing code at build time.
  - Not run in the simulator!
  - “Clean room” & granular

- Application Tests
  - Testing (dynamic) code at run time.
  - Can test hardware (e.g., location) too!
OS X: Xcode 3 Project Setup

- Create a new target to run the tests.
  - Add: “Unit Test Bundle”
- Add the OCUnit SenTestingKit Framework
- Create a new group for the test cases.
  - Just a group, it’s not a folder.
- Create new test case(s)
  - Add: “Objective-C test case class”
- Build target, see errors in Debugger:Console
OS X: Xcode 3 Project Setup

- **Independent**
  - **Pro:** Easier to set up. Run only when wanted.
  - **Con:** Must be run manually, which can lead to infrequent use.

- **Dependent**
  - **Pro:** Easier to run tests automatically. Still allows option to build main executable separately.
  - **Con:** Require some additional steps to set up.
  - **Set up as a “direct dependent.”**
iOS & Xcode 3: Project Setup

- Both tests are also (new) targets, but the “App tests” are interdependent with the application you are testing.
iOS & Xcode 3: Logic Test Setup

• Same as you did for OS X applications
• Be sure you use iOS, and not Mac OS X!
iOS & Xcode 3: App. Testing Setup

- Make a copy of the application
- Add the iOS unit testing bundle target
- Set up dependency
  - Bundle product to App’s Copy phase
- Only runs on the device!
  - Output appears in Debugger Console
Testing Harness

• Test Case Class (Interface)
  • Inherits from “SenTestCase”
  • Methods are the test cases

• Adding an “Objective-C test case class” provides you with a template:

```objective-c
#import <SenTestingKit/SenTestingKit.h>

@interface MyTests : SenTestCase
{
}
@end
```
Writing Test Case Methods

- Test case names begin with the word “test”
- Test cases take no parameters
- Test cases’ return type is (void)

In the @interface:

```objective-c
@interface MyTests : XCTestCase
{
}
@end

-(void) test1;
```
Writing Test Cases

• In the `@implementation`:

```c
- (void) testCase1
{
    ...
}
```
Some Handy Macros

STAssertNotNil(a1, description, ...)

STAssertTrue(expression, description, ...)

STAssertFalse(expression, description, ...)

STAssertEqualObjects(a1, a2, description, ...)

STAssertEquals(a1, a2, description, ...)

STAssertThrows(expression, description, ...)

STAssertNoThrow(expression, description, ...)

STFail(description, ...)

...and many, many more. There is a copy of the full macro reference in the lab (under “OCUnit Macros”).
Automating UI Events

• Quite a bit more involved
• UI Instruments “Automation” tool

• Needs a javascript test file:

```javascript
// Get the handle of applications main window
var window = UIATarget.localTarget().frontMostApp().mainWindow();

// Get the handle of view
var view = window.elements()[0];

var textfields = window.textFields();
...

// Check number of Text field(s)
if(textfields.length!=1)
{
    UIALogger.logFail("FAIL: Inavlid number of Text field(s)");
}
else
{
    UIALogger.logPass("PASS: Correct number of Text field(s)");
}
```
Automating UI Events

- Apple
  - UI Automation Reference Collection
  - Instruments: Built-In Instruments: UI Automation
- O’Reilly Books
  - Tutorial on the lab’s “additional information” page.
"Writing Testable Code"

- Define API requirements.
  - Inputs, outputs, and exceptions
- Write test cases as you write code.
  - Test-driven Development (Agile)
- Check boundary conditions.
  - Test the edges of acceptable inputs
- Use negative tests.
  - Correctly handle bad inputs
  - Check your exceptions
“Writing Testable Code” (continued)

- Write comprehensive test cases.
  - Layer tests
    - e.g., Create an array, fill it with objects, remove a few, ensure that the array’s contents are what you expect.
- Cover your bug fixes with test cases.
  - When you fix a bug, write a test case for it!
Today’s “Lab”

• Uses Apple’s step by step instructions.
• See associated notes first.
• Download projects if you have trouble finding them in the docs.
• Full STxxx() macro reference.
• Additional information (links), including automated UI testing.
• Homework: end of next Friday (30th)