

Automated Testing with Objective-C

Ben Scheirman

Ben Scheirman

Director of Development at ChaiOne



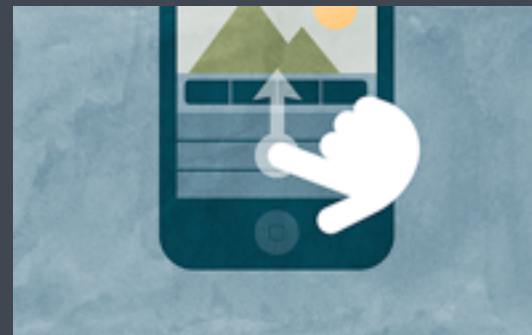
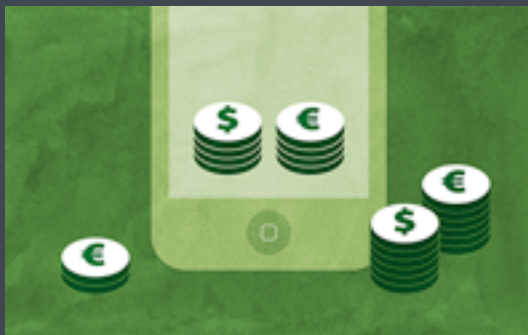
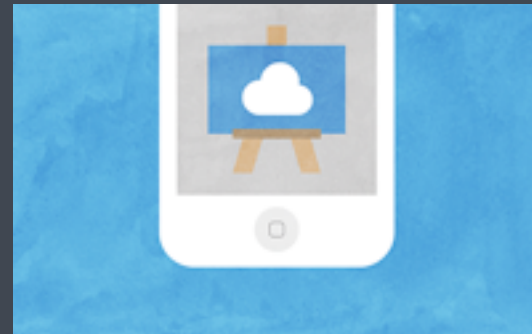
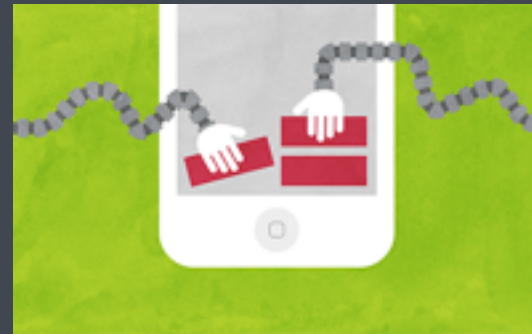
Houston, Texas, USA

@subdigital
benscheirman.com

ChaiOne 



NS Screencast



nsscreencast.com

Automated Tests

Automated Tests



**Does your
application work?**



Automated Tests



Application
Code

Automated Tests



Application
Code



Automated Tests



Application
Code

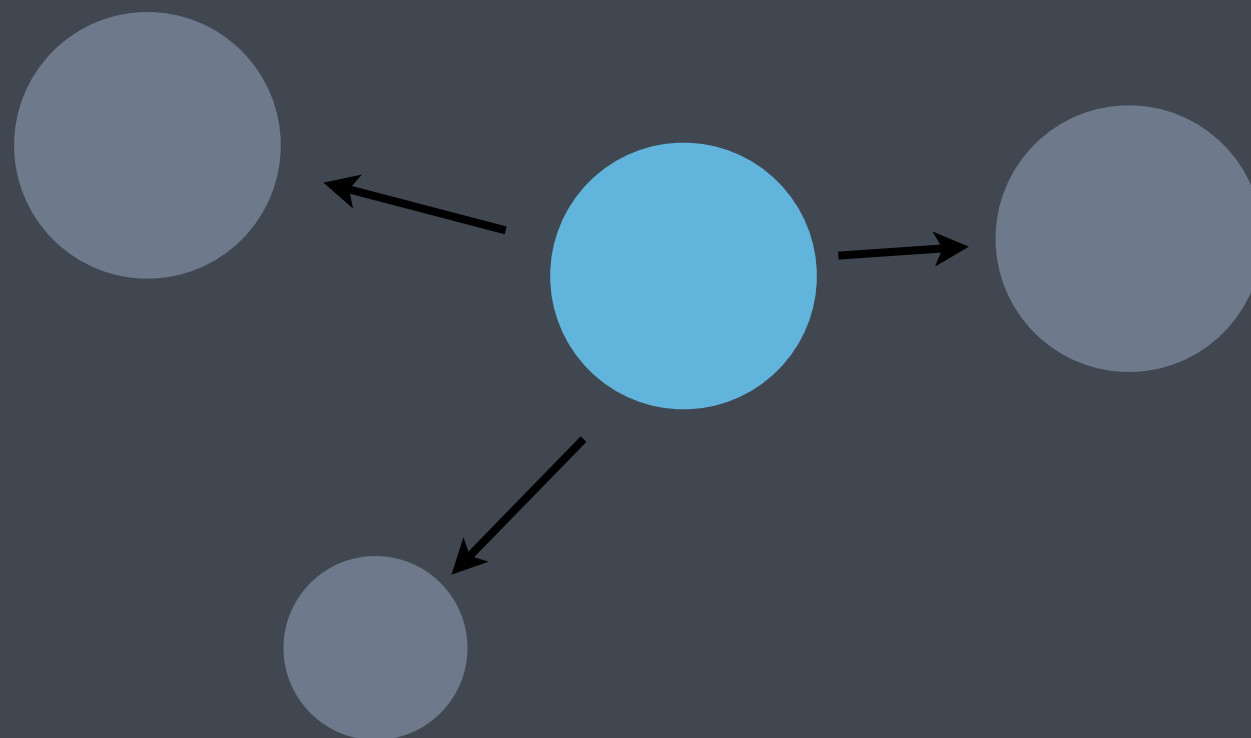


Unit Tests

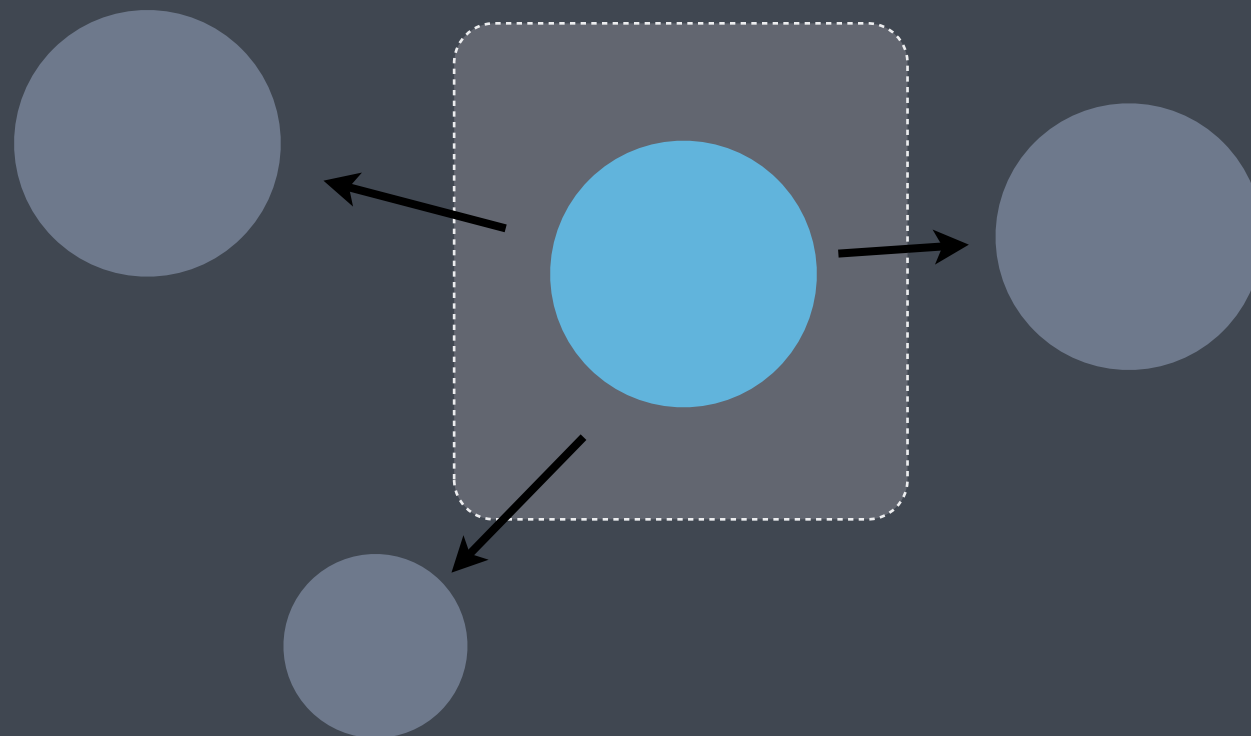
Integration Tests

Acceptance Tests

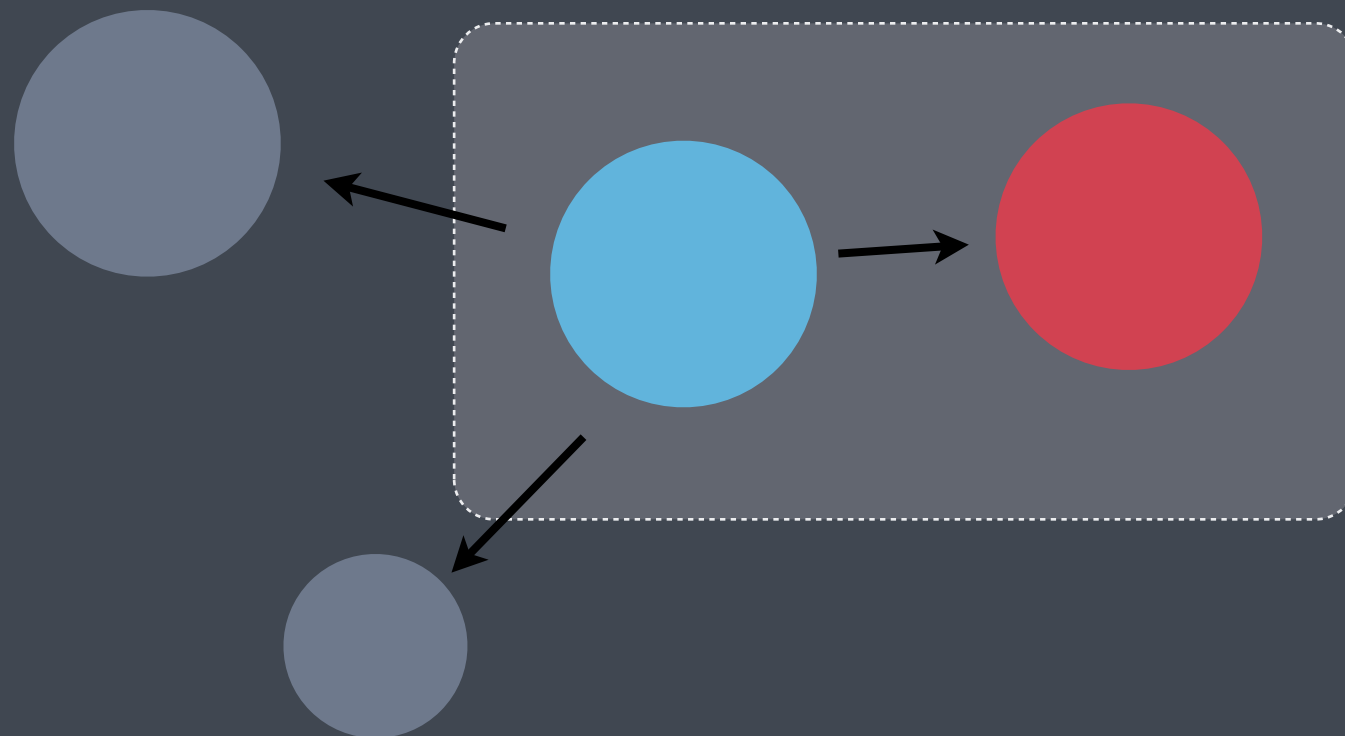
Unit Tests



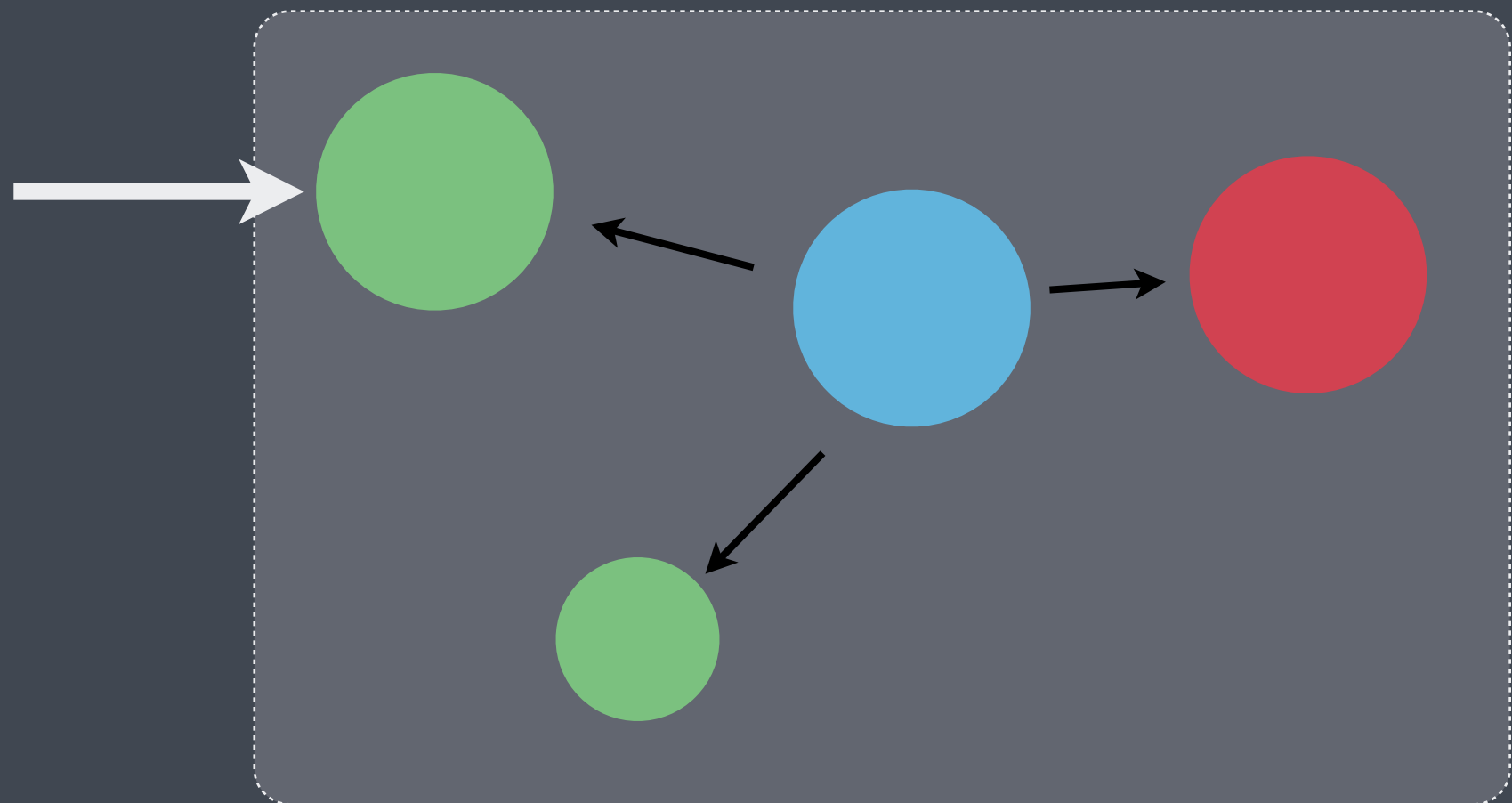
Unit Tests



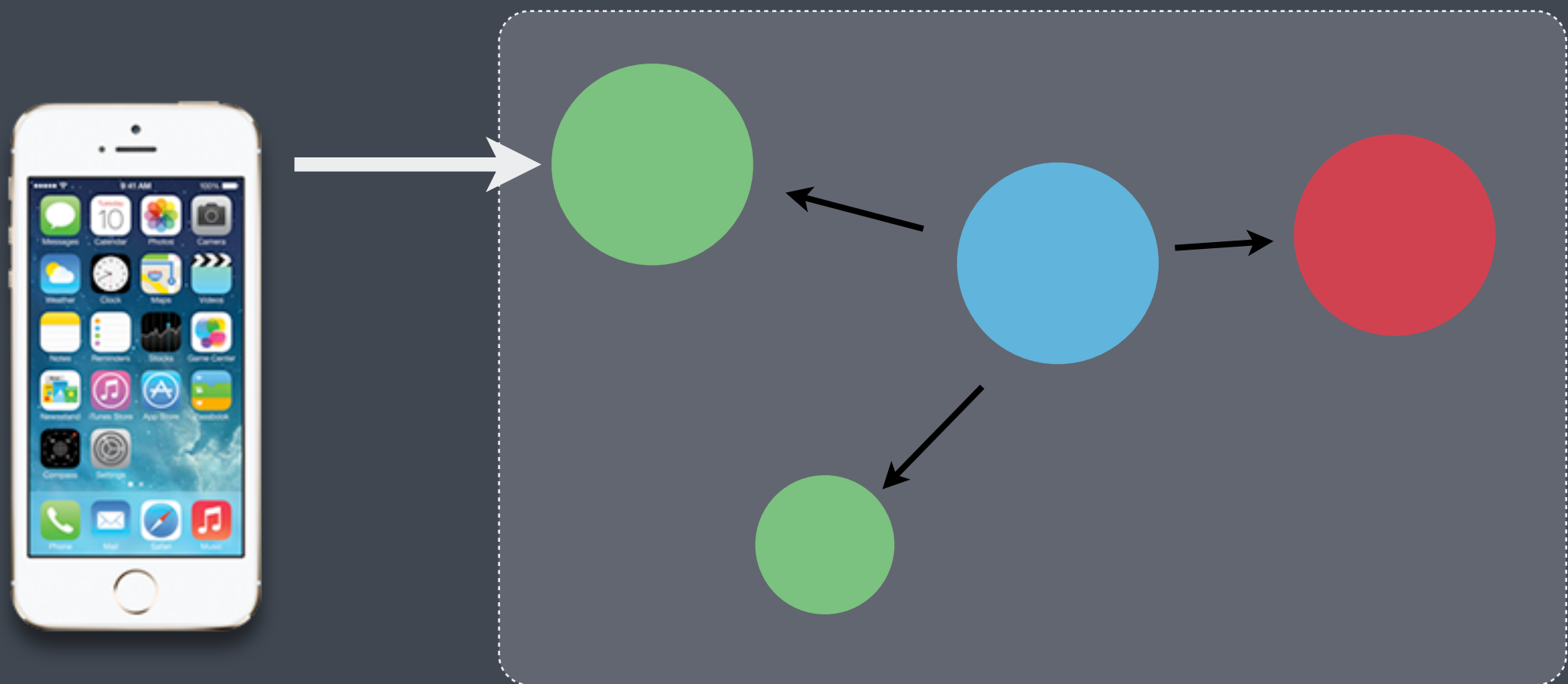
Integration Tests



Acceptance Tests



Acceptance Tests







What's in a test run?

What's in a test run?

Setup

Test

Tear Down

Setup

Test

Tear Down

What's in a test?

Arrange

Act

Assert

Test

What's in a test?



Test

What's in a test?

Test

```
- (void)test_adding_numbers {  
    Calculator *calc = [Calculator new];  
    int sum = [calc add:x to:y];  
  
    XCTAssertEqual(7, sum,  
        @"math is hard :(");  
}
```

What's in a test?

Test

```
- (void)test_adding_numbers {
```

```
    Calculator *calc = [Calculator new];
```

```
    int sum = [calc add:x to:y];
```

```
    XCTAssertEqual(7, sum,  
        @"math is hard :(");
```

```
}
```

Arrange



What's in a test?

Test

```
- (void)test_adding_numbers {  
    Calculator *calc = [Calculator new];  
    int sum = [calc add:x to:y];  
  
    XCTAssertEqual(7, sum,  
        @"math is hard :(");  
}
```

Act



What's in a test?

Test

```
- (void)test_adding_numbers {  
    Calculator *calc = [Calculator new];  
    int sum = [calc add:x to:y];
```

```
    XCTAssertEquals(7, sum,  
        @"math is hard :(");
```

```
}
```

Assert



1 assertion
per
test



Underline lines
& points

Division control
over font, font
style, & paragraph
settings

Writing
style

Color palette

Transparencies

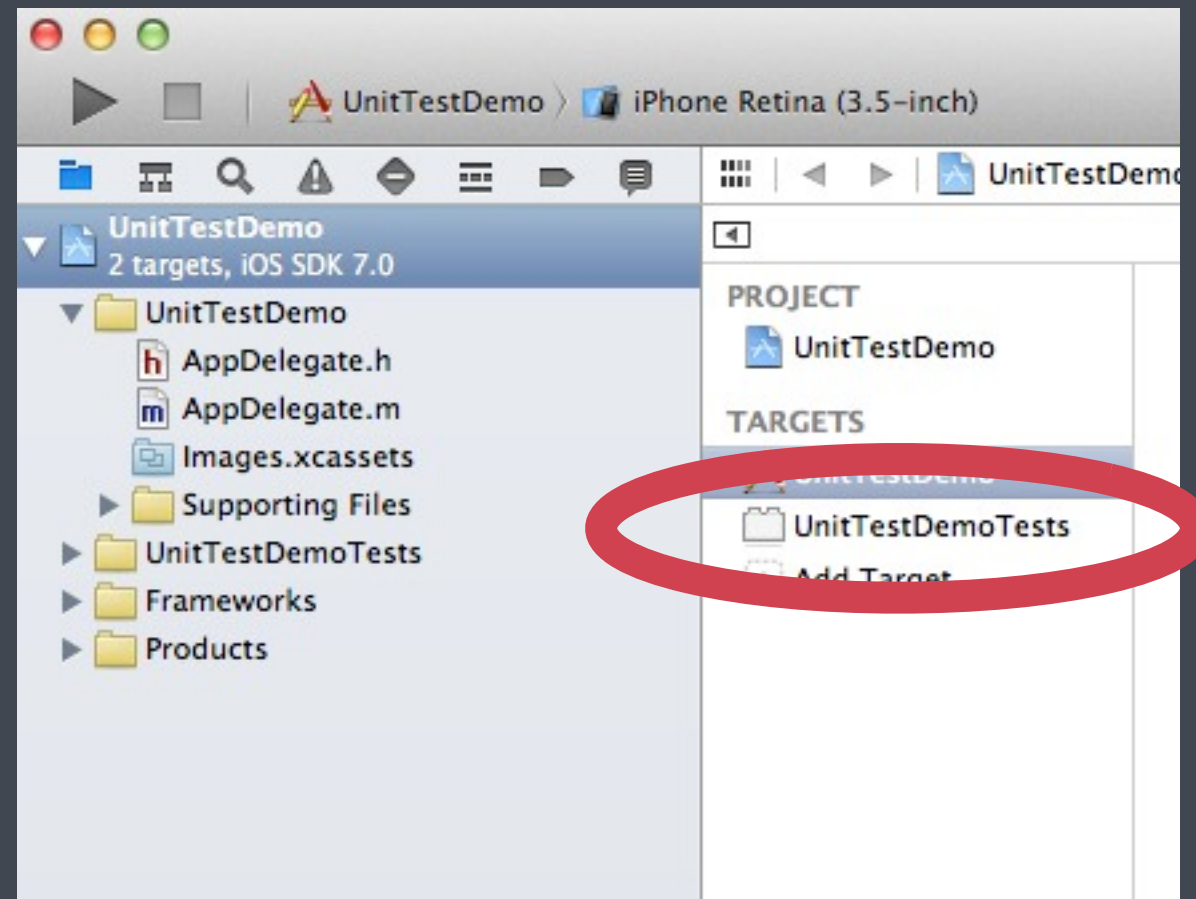
Writing rules
- Font spacing
- Alignment
- Color text



PROJECT: APPLICATION APP

VIEW

PROJECT: XCODE #1



Unit Tests come **default** on
Xcode 5









BoardGameTests
31 tests, 2 failing



BoardTests



testUniqueLocations





testInvalidMoves





testPieceReplacement





testCheatMode





KingMoveTests





LocationTests



testNegativeLocation



testNaNLocation



testLocationMutability



MoveTests



testOutOfBoundsMove



testDiagonalMove



testHorizontalMove



testVerticalMove



testForwardMove



testBackwardMove



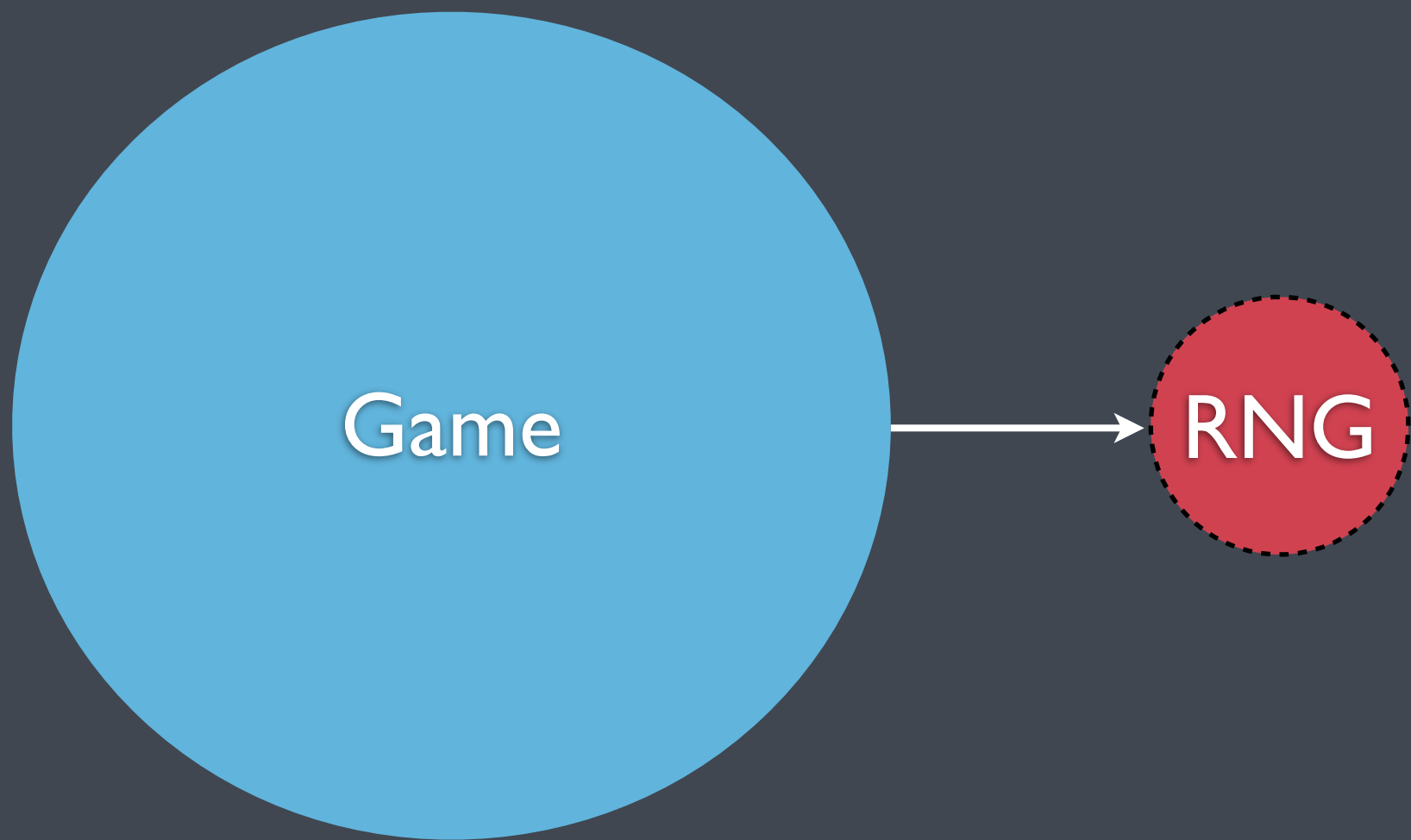
Demo: Guess the number game

**How do we know
what number it will
generate?**

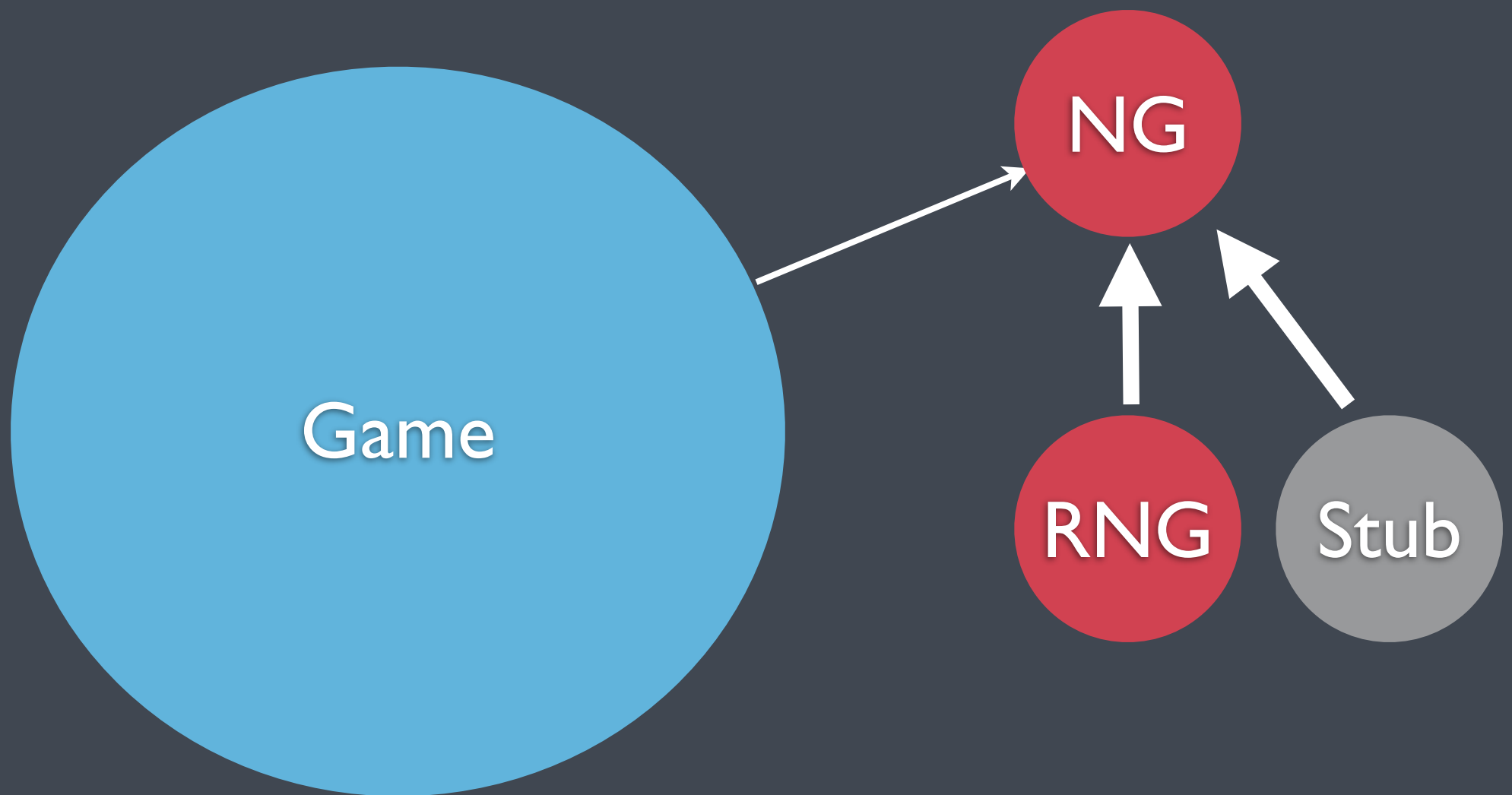


Game









Demo: Stubbing with OCMock

What about...

View Controllers?

Interface Builder?

Storyboards?

DETOUR AHEAD

Behavior Driven Development

GIVEN WHEN THEN

GIVEN WHEN THEN

Some Behavior

GIVEN WHEN THEN

Some Behavior

I do X

GIVEN WHEN THEN

Some Behavior

I do X

Y should happen

Specta / Expecta

```
SpecBegin( CalculatorSpec)
```

```
describe(@"Calculator", ^{  
  __block Calculator *_calc;  
  beforeEach(^{  
    _calc = [Calculator new];  
  });
```

```
    it(@"should add two numbers", ^{  
      int sum = [_calc add:5 to:7];  
      expect(sum).to.equal(12);  
    });  
  });
```

```
SpecEnd
```

Expecta

```
expect(sum).to.equal(12);
```

Kiwi

```
[theValue(sum) should] equal:theValue(12);
```

Expecta

Leverage OCMock for mocks & stubs

Kiwi

Built-in powerful mocking & stubbing
functionality

Testing Storyboards

```
UINavigationController *mainStoryboard = [UINavigationController  
    storyboardWithName:@"MainStoryboard" bundle:nil];  
  
UINavigationController *nav = [mainStoryboard  
    instantiateInitialViewController];  
  
ViewController *vc = (ViewController *)[nav visibleViewController];  
  
UIView *view = vc.view;  
expect(view).toNot.beNil();
```

Testing Wired Actions

```
UIButton *button = _vc.loginButton;  
NSArray *actions = [button  
    actionsForTarget:_vc  
    forControlEvents:UIControlEventTouchUpInside];  
  
expect(actions[0]).to.equal(@"loginTapped:");
```

Testing Block Based interaction

VC

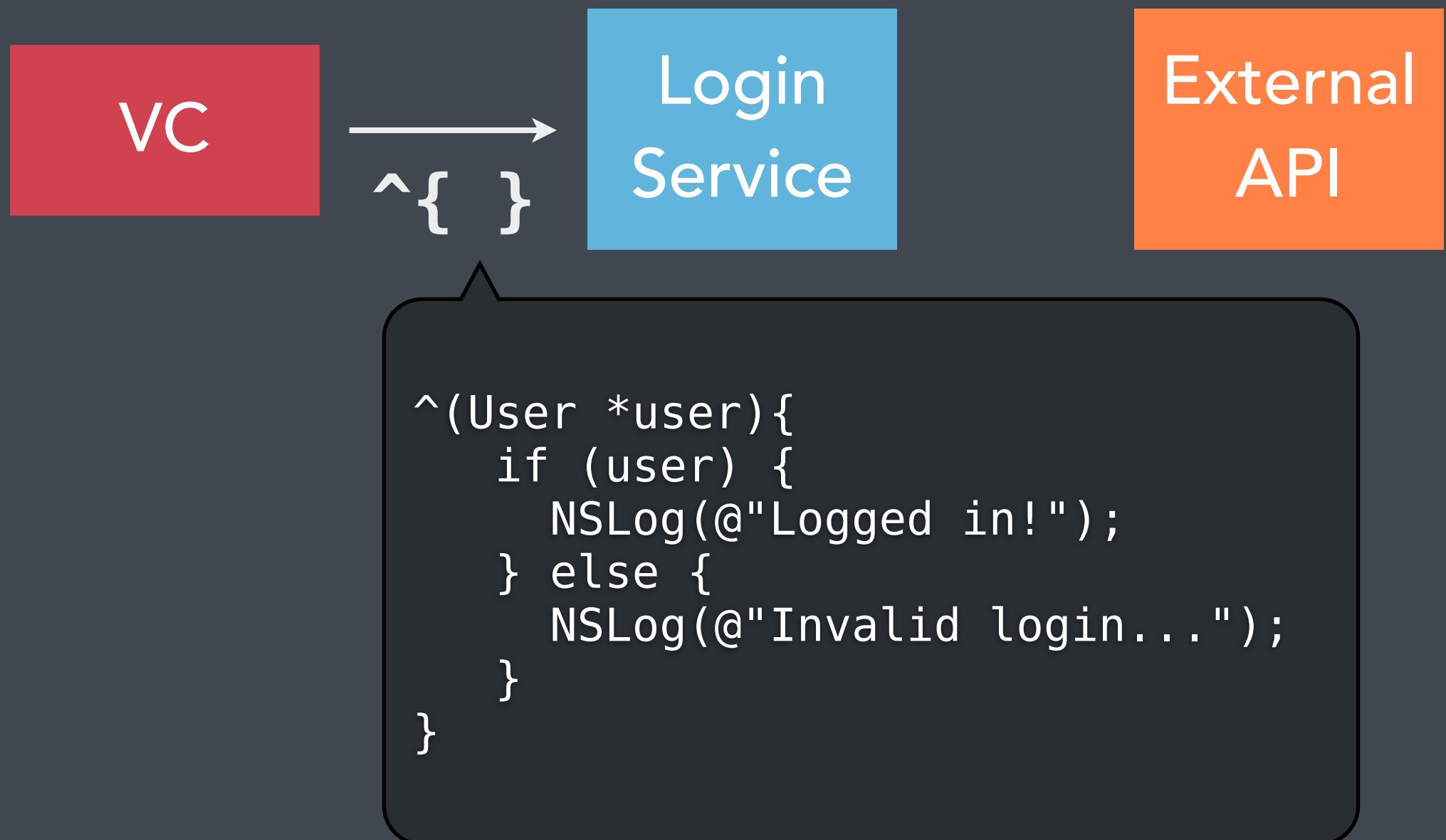
Login
Service

External
API

Testing Block Based interaction



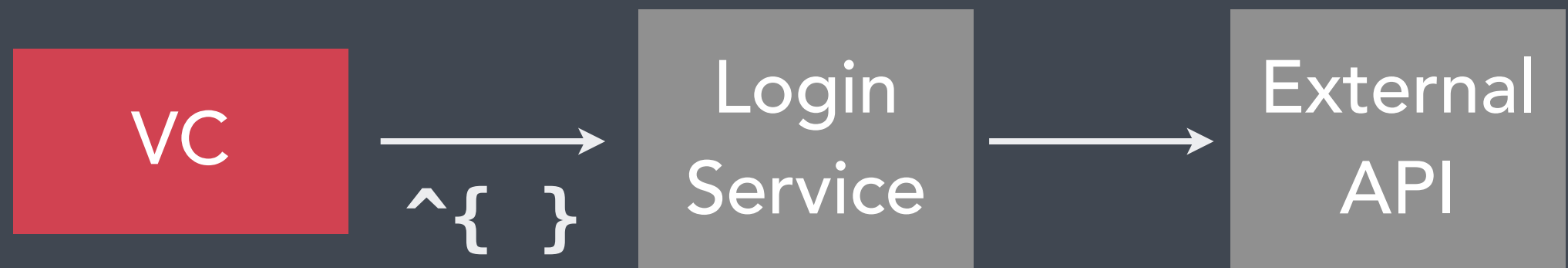
Testing Block Based interaction

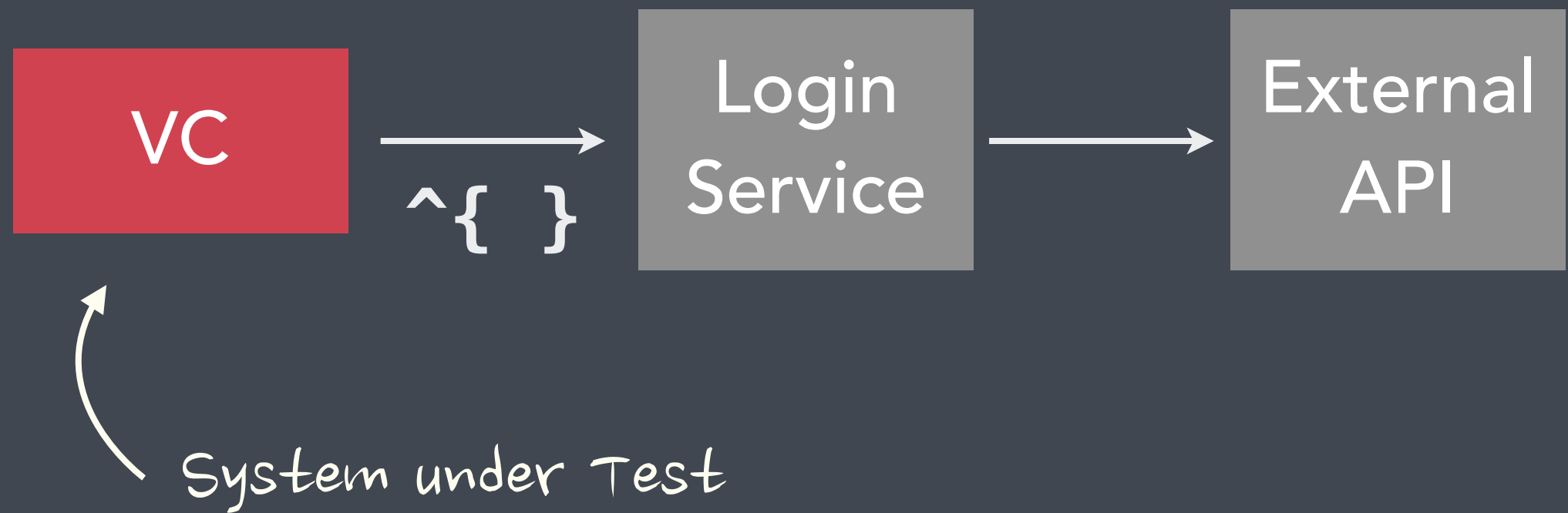


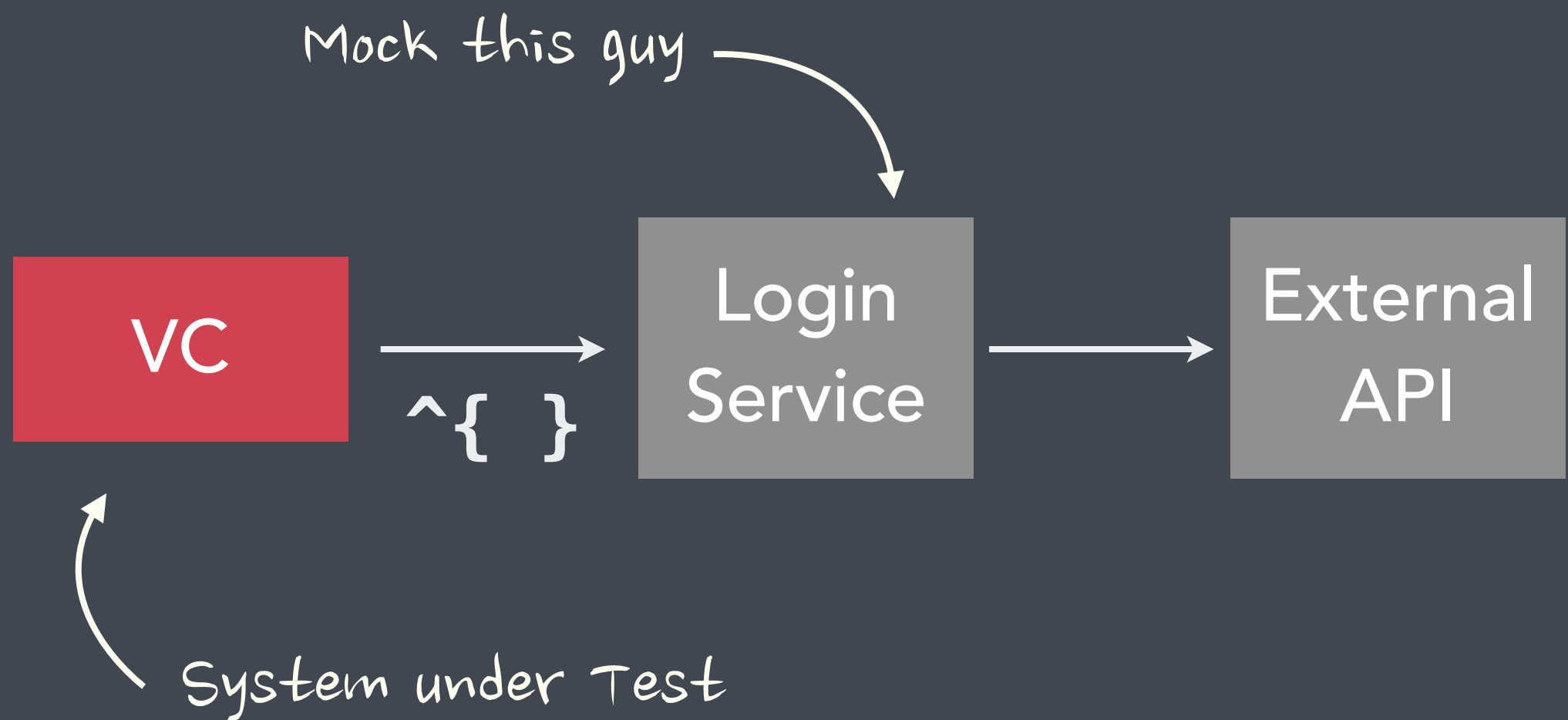
Testing Block Based interaction



```
^(User *user){  
    if (user) {  
        NSLog(@"Logged in!");  
    } else {  
        NSLog(@"Invalid login...");  
    }  
}
```







**Step 1: Verify Login
Service is called**

```
describe(@"logging in", ^{
    it(@"should verify username & password w/ login svc", ^{
        id mockLoginService =
            [OCMockObject mockForClass:[LoginService class]];

        [[mockLoginService expect] verifyUsername:@"user1"
                                         password:@"password1"
                                         completion:
                                             [OCMArg any]];

        _vc.loginService = mockLoginService;
        _vc.usernameTextField.text = @"user1";
        _vc.passwordTextField.text = @"password1";

        [_vc loginTapped:nil];

        [mockLoginService verify];
    });
});
```

```
describe(@"logging in", ^{
    it(@"should verify username & password w/ login svc", ^{
        id mockLoginService =
            [OCMockObject mockForClass:[LoginService class]];

        [[mockLoginService expect] verifyUsername:@"user1"
                                         password:@"password1"
                                         completion:
                                         [OCMArg any]];

        _vc.loginService = mockLoginService;
        _vc.usernameTextField.text = @"user1";
        _vc.passwordTextField.text = @"password1";

        [_vc loginTapped:nil];

        [mockLoginService verify];
    });
});
```

**Step 2: Define the
behavior when the login
service calls back**

Step 2: Define the behavior when the login service calls back

2 cases: success & failure

```
describe(@"logging in", ^{
  it(@"should welcome the user when the login is valid", ^{
    id mockLoginService =
      [OCMockObject mockForClass:[LoginService class]];

    [[mockLoginService expect] verifyUsername:[OCMAArg any]
                                     password:[OCMAArg any]
                                     completion:...
                                     ];

    _vc.loginService = mockLoginService;
    [_vc loginTapped:nil];

    [mockLoginService verify];
  });
});
```

```
describe(@"logging in", ^{
  it(@"should welcome the user when the login is valid", ^{
    id mockLoginService =
      [OCMockObject mockForClass:[LoginService class]];

    [[mockLoginService expect] verifyUsername:[OCMArg any]
                                           password:[OCMArg any]
                                           completion:...
                                           ];

    _vc.loginService = mockLoginService;
    [_vc loginTapped:nil];

    [mockLoginService verify];
  });
});
```

```
[[mockLoginService expect] verifyUsername:[OCMAArg any]  
                                password:[OCMAArg any]  
                                completion:[OCMAArg  
checkWithBlock:^(BOOL(id completionBlock) {  
    // fake login successful for a user  
    id mockUser = [OCMockObject mockForClass:[User class]];  
    completionBlock(mockUser);  
    return YES; // signify that this argument is valid  
}]];
```

Where to go from here?

Explore Acceptance Test Frameworks:

KIF

Frank

UIAutomation

Some Parting Tips...

If it's hard to test, improve your design

Some Parting Tips...

If it's hard to test, improve your design

Try to test 1 thing

Some Parting Tips...

If it's hard to test, improve your design

Try to test 1 thing

**Write tests first to avoid creating
untestable designs**

Some Parting Tips...

If it's hard to test, improve your design

Try to test 1 thing

**Write tests first to avoid creating
untestable designs**

Use a continuous integration server

Some Parting Tips...

If it's hard to test, improve your design

Try to test 1 thing

**Write tests first to avoid creating
untestable designs**

Use a continuous integration server

PRACTICE!

Thank you!

Ben Scheirman

@subdigital

benscheirman.com