Overview

Writing maintainable tests
About Chris

- Grew up in England and live in Oakland, CA
- Over twenty years of software development experience
  - Building object-oriented software since 1986
  - Using Java since 1996
  - Using J2EE since 1999
- Author of POJOs in Action
- Speaker at JavaOne, SpringOne, NFJS, JavaPolis, Spring Experience, etc.
- Chair of the eBIG Java SIG in Oakland (www.ebig.org)
- Run a consulting and training company that helps organizations build better software faster and deploy it on Amazon EC2
- Founder of Cloud Tools, an open-source project for deploying Java applications on Amazon EC2: http://code.google.com/p/cloudtools
Agenda

- Tests - a double-edged sword
- Taming test fixture logic
- Simplifying verification code
- Writing web tests
- Testing Ajax applications
Why test?

- Write new code more easily
  - Automates what we are doing already – Right!?
  - Run fast unit tests instead of slower web application
  - Use TDD to incrementally solve a problem
- Tests are a safety net
  - Confidently change existing code
  - Easier to refactor code to prevent decay
- Fewer bugs that impact customers AND development
Why test?

• Increases longevity:
  ➢ Testable code is cleaner code
  ➢ Without tests your application will decay and die

• Absolutely essential when using a dynamic language
  ➢ Compiler can't catch typos
  ➢ Nothing is too simple to test
  ➢ You need unit tests
### 4 phase tests

- **Setup**
- **Exercise**
- **Verify**
- **Teardown**

```java
public class FooTest
    extends TestCase {

    public void setUp() {
    }

    public void testSomething() {
        // test-specific setup

        // Exercise the SUT

        // Verification

        // test-specific tear down
    }

    public void tearDown() {
    }

}
Types of tests

- **Domain model tests**
  - Test your domain objects and services
  - In-memory tests

- **Persistence tests**
  - Test manipulating persistent objects

- **Service integration tests**
  - Test services using database

- **Web tests**
  - Automatic tests using Selenium
  - Click and type in the GUI
Example tests

- Walk through some example ptrack tests
The trouble with tests

- They make software more difficult to change
  - That's a good thing since they detect bugs
- But change is inevitable: new features, refactoring
- If you can't easily change the test code:
  - Slows down the development
  - Tests are removed or abandoned
Poor quality test code

- Common test code smells
  - Obscure Tests – you can't tell what a test does
  - Test code duplication – copy and paste tests
- Badly structured test setup logic
  - Complicated logic to create test fixtures
  - *e.g.* the test objects (in-memory or in DB)
- Sprawling web tests
  - Web test framework APIs are very low-level.
  - Easily end up with large amounts of difficult to maintain code: `click()`, `type()`, ...
  - Lots of duplication
  - Lots of obscure code
Excellent testing book

- Test smells and how to fix them
  - Obscure test
  - Fragile test
  - Test code duplication
  - ...

- Comprehensive pattern language:
  - Four phase test
  - Minimal fixture
  - Test utility method
  - Test helper
  - Humble Object
  - ...

© Copyright 2008, Chris Richardson
Agenda

- Tests – a double-edged sword
- Taming test fixture logic
- Simplifying verification code
- Writing web tests
- Testing Ajax applications
What's a fixture

- **Fixture** = everything that needs to be in place to test an object/system
  - Object/system we want to test
  - Its collaborators, required test data, *etc.*

- Fixture is created by **test fixture logic**:
  - Code in the test methods themselves
  - JUnit 3.x `setUp()`
  - JUnit 4/TestNG `@Before*` annotations
The challenge of test fixtures

- Creating an object isn't always easy
  - Objects can have lots of attributes
  - Objects are often aggregate roots
  - `new()` is often insufficient

- Test fixtures often create multiple objects
  - *e.g.* money transfer test needs two accounts

- Multiple tests need the same set of objects
  - `AccountTests, MoneyTransferServiceTests` need similar `Account` objects
Object graphs can be complicated

We need all of these objects to test a Project!
Constructing individual objects can be tricky

- Best way to construct an object is to use a non-default constructor:
  - Supports objects without setters
  - Supports immutable objects
  - Forces you to supply all required objects
  - Constructor can verify object state

- Limitations of constructors:
  - Lots of constructor arguments ⇒ code is difficult to read
  - Optional arguments ⇒ multiple constructors
The alternative to constructors

Project project = new Project(initialStatus);
   project.setName("Test Project #1");
   project.setDescription("description");
   project.setRequirementsContactName("Rick Jones");
   project.setRequirementsContactEmail("jones@nowhere.com");
   project.setType(ProjectType.EXTERNAL_DB);
   project.setArtifacts(new ArtifactType[] {
       ArtifactType.ARCHITECTURE,
       ArtifactType.DEPLOYMENT, ArtifactType.MAINTENANCE
   });
   project.setInitiatedBy(user);

Benefits:
  • Handles optional parameters
  • Is more readable

But
  • Lots of noise: 'project.set'
  • Breaks encapsulation
  • Object is mutable
  • Cannot validate state
Constructing objects fluently

```java
Project project = new Project(initialStatus)
    .name("Test Project #1")
    .description("Excellent project")
    .initiatedBy(user)
    .requirementsContactName("Rick Jones")
    .requirementsContactEmail("jones@nowhere.com")
    .type(ProjectType.EXTERNAL_DB)
    .artifact(ArtifactType.ARCHITECTURE)
    .artifact(ArtifactType.DEPLOYMENT)
    .artifact(ArtifactType.MAINTENANCE);
```

**Chained method calls**

**Benefits:**
- Less noise
- Meaning of each value is clear

**Drawbacks:**
- Breaks encapsulation – objects must have mutators/setters
- Doesn't work with immutable objects
- No opportunity to validate state
Fluently creating immutable objects

- Initialize the mutable builder
- `make()` instantiates the domain object via a constructor
- Allows entity to be immutable
- Builder can validate object state

This is an internal DSL

```java
Project project = new Project.ProjectBuilder(initialStatus)
    .name("Test Project #1")
    .description("description")
    .initiatedBy(user)
    .requirementsContactName("Rick Jones")
    .requirementsContactEmail("jones@nowhere.com")
    .type(ProjectType.EXTERNAL_DB)
    .artifact(ArtifactType.ARCHITECTURE)
    .artifact(ArtifactType.DEPLOYMENT)
    .artifact(ArtifactType.MAINTENANCE)
    .make();
```

Nested Entity Builder example

class Project {

    public static class ProjectBuilder {
        private String name;
        ...

        public ProjectBuilder(Status initialStatus) {
            this.status = status;
        }

        ProjectBuilder name(String name) {
            this.name = name;
            return this;
        }

        public Project make() {
            // Verify that we have everything
            return new Project(this);
        }
    }

    public Project(ProjectBuilder builder) {
        this.name = builder.name;
        this.initiatedBy = builder.initiatedBy;
    }

    • Pass builder as the sole constructor argument
Testing makes your objects smarter

- Production code often has simple requirements:
  - Create using default constructor
  - Accesses Java bean properties
- But tests need smarter objects
  - e.g. fluent interfaces
  - Counter to the concept of not having test code in production code
- But this is ok: tests are important
Centralizing test object creation with Object Mothers

- Fluent interfaces can help but
  - Fixture logic can still be complex
  - Same object created in multiple tests ⇒ duplication
- Eliminate duplication:
  - Centralize object creation in a test utility class called an Object Mother
  - Defines factory methods for creating fully formed aggregate
  - Different methods for different aggregate states

```java
public class ProjectMother {
    public static Project makeProjectInProposalState (                Status initialStatus, User user) {
        ...
    }

    public static Project makeProjectInRequirementsState(
        Status initialStatus, User user) {
        ...
    }
}
```

Creating multiple connected aggregates

- Each Object Mother method creates a single aggregate
- But some tests need to create multiple aggregates with shared sub-aggregates
- Must avoid duplicating that code in multiple tests

```java
itDepartment = DepartmentMother.makeItDepartment();

itProjectManager = UserMother.makeProjectManager(itDepartment);

itBusinessAnalyst = UserMother.makeBusinessAnalyst(itDepartment);

projectInCompleteState = ProjectMother.makeProjectInCompleteState(...);

projectInRequirementsState = ProjectMother.makeProjectInRequirementsState(...);

...
Use stateful object mothers

- Test instantiates object mother
- Object mother's constructor
  - Creates aggregates (by calling their Object Mothers)
  - Stores them in (public) fields
- Test gets the data it needs from the object mother
Example of a stateful mother

```java
public class PTrackWorld {

    private final Department itDepartment;
    private final User itProjectManager;
    private final User itBusinessAnalyst;
    private final Project projectInCompleteState;

    ...

    public PTrackWorld() {
        itDepartment = DepartmentMother.makeItDepartment();

        itProjectManager = UserMother.makeProjectManager(itDepartment);
        itBusinessAnalyst = UserMother.makeBusinessAnalyst(itDepartment)

        stateMachine = DefaultStateMachineFactory.makeStateMachine("default");
        initialStatus = stateMachine.getInitialStatus();
        projectInCompleteState = ProjectMother.makeProjectInCompleteState(initialStatus,
                itProjectManager, getAllITDepartmentEmployees());

        ...
    }
}
```

```java
public class ProjectTests extends TestCase {

    private Project project;
    private User projectManager;
    private User businessAnalyst;

    protected void setUp() throws Exception {
        PTrackWorld world = new PTrackWorld();
        projectManager = world.getItProjectManager();
        businessAnalyst = world.getItBusinessAnalyst();
        project = world.getProjectInProposalState();
    }
}
```
Object Mothers and databases

- Initialize database using objects created by mothers
  - Create objects using mothers
  - Persist them
- Very easy when using ORM
- Avoids difficult to maintain flat files: CSV, SQL, XML

```java
public class PtrackDatabaseInitializer
    implements InitializingBean, DatabaseInitializer {

    private HibernateTemplate template;
    private PTrackWorld world;

    public PtrackDatabaseInitializer(HibernateTemplate template) {
        this.template = template;
    }

    public void afterPropertiesSet() {
        initializeDatabase();
    }

    public void initializeDatabase() {
        world = new PTrackWorld();
        StateMachine stateMachine = world.getStateMachine();
        template.save(stateMachine);
        Department itDepartment = world.getITDepartment();
        template.save(itDepartment);
        ...
    }
}
```
Object Mother design

- **Choices**
  - Same data each time *vs.* random data
  - Referenced aggregates as parameters *vs.* create those aggregates too

- **Tip:** use descriptive names
  - Bad: makeProject1(), makeProject2(), ...
  - Better: makeProjectIn&lt;State&gt;(), ...

- **Tip:** use Object Mothers from day 1
Finding balance

public void testOptionA() {
    ShoppingCart cart
        = ShoppingCartMother.makeEmptyCart();
    cart.add(ProductMother.makeInstockPart(), 1);
    cart.add(ProductMother.makeBackOrderdPart(), 1);
    cart.add(ProductMother.makeDiscontinuedPart(), 1);
    ...
}

OR

public void testOptionB() {
    ShoppingCart cart =

    ShoppingCartMother.makeWithInstockPartBackorderedPartandDiscontinuedPart();
    ...
}

Intention revealing
But risks duplication

Ridiculously long names???
Agenda

- Tests – a double-edged sword
- Taming test fixture logic
- **Simplifying verification code**
- Writing web tests
- Testing Ajax applications
Writing readable verification logic

- Verification phase verifies that expected outcome has been obtained
- State verification makes assertions about:
  - Returned value
  - State of system under test
  - State of collaborators
- Test frameworks provide the basic assert methods but we must:
  - Ensure readability
  - Avoid code duplication
protected void setUp() throws Exception {
    PTrackWorld world = new PTrackWorld();
    projectManager = world.getItProjectManager();

    project = world.getProjectInProposalState();
    startTime = new Date();

    state0 = world.getInitialState();
    state1 = state0.getApprovalStatus();
    state2 = state1.getApprovalStatus();
}

public void testChangeStatus() throws InterruptedException {
    boolean result = project.changeStatus(true, projectManager, "Excellent");
    Date endTime = new Date();

    assertTrue(result);
    assertEquals(state1, project.getStatus());
    assertEquals(1, project.getHistory().size());
    Operation operation = project.getHistory().get(0);
    assertEquals("Excellent", operation.getComments());
    assertEquals(projectManager, operation.getUser());
    ...  
    assertFalse(operation.getTimestamp().before(startTime));
    assertFalse(operation.getTimestamp().after(endTime));
}
Using Custom Assertions

- Verification code calls a Test Utility Method that makes assertions
- Has an Intention Revealing Name
- Benefits:
  - Makes the code more readable
  - Eliminates duplication
Simplified test

```java
protected void setUp() throws Exception {
    ...
    expectedOperation0 =
        new Operation(null, projectManager,
                      state0, state1, 
                      "Excellent");
}

public void testChangeStatus() {
    boolean result = project.changeStatus(true, 
                                            projectManager, "Excellent");
    Date endTime = new Date();

    assertTrue(result);
    assertEquals(state1, project.getStatus());
    assertHistoryContains(project, startTime, endTime,
                           expectedOperation0);
}
```
public void assertOperationEqual(Operation expectedOperation, Date startTime, Date endTime, Operation operation) {
    assertEquals(expectedOperation.getComments(), operation.getComments());
    ... operation.getUser());
    assertEquals(expectedOperation.getFromStatus(), operation.getToStatus());
    assertFalse(operation.getTimestamp().before(startTime));
    assertFalse(operation.getTimestamp().after(endTime));
}

private void assertHistoryContains(Project project, Date startTime, Date endTime, ... expectedOperations) {
    int i = 0;
    List<Operation> history = project.getHistory();
    assertEquals(expectedOperations.length, history.size());
    for (Operation expectedOperation : expectedOperations) {
        Operation operation = history.get(i++);
        assertOperationEqual(expectedOperation, startTime, endTime, operation);
        startTime = operation.getTimestamp();
    }
}

private void assertOperationEqual(Operation expectedOperation, Date startTime, Date endTime, Operation operation) {
    assertEquals(expectedOperation.getComments(), operation.getComments());
    assertEquals(expectedOperation.getUser(), operation.getUser());
    assertEquals(expectedOperation.getFromStatus(), operation.getFromStatus());
    assertEquals(expectedOperation.getToStatus(), operation.getToStatus());
    assertFalse(operation.getTimestamp().before(startTime));
    assertFalse(operation.getTimestamp().after(endTime));
}
Literate assertions with Hamcrest

- Hamcrest is an open-source framework
- Readable "literate" assertions
- Rich set of composable matchers
- Literate error messages
- Used by Jmock expectations

```java
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.hamcrest Matchers.isOneOf;

assertThat(project.getStatus(), is(state1));
assertThat(project.getStatus(), isOneOf(state1, state2));
assertThat(project.getStatus(), allOf(is(state), not(is(state2))));
```
public class ProjectMatchers {

    public static Matcher<Date> withinInclusivePeriod(final Date startTime, final Date endTime) {
        return new TypeSafeMatcher<Date>() {
            public boolean matchesSafely(Date date) {
                return !date.before(startTime) && !date.after(endTime);
            }

            public void describeTo(Description description) {
                description.appendText(String.format("expected to be between <%s> and <%s>", startTime, endTime));
            }
        };
    }

    import static org.jia.ptrack.domain.ProjectMatchers.withinInclusivePeriod;

    public void testChangeStatus() {
        assertThat(operation.getTimestamp(), is(withinInclusivePeriod(startTime, endTime)));
    }
}

java.lang.AssertionError:
  got: "Wed Dec 31 16:00:00 PST 1969"
  at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:21)
Agenda

- Tests - a double-edged sword
- Taming test fixture logic
- Simplifying verification code
- **Writing web tests**
- Testing Ajax applications
Writing web tests

- Web tests simulate the user
  - Fill-in forms
  - Click buttons and links
- Assertions:
  - Correct page displayed
  - Correct data is displayed
  - Page elements exist/visible
Using Selenium

- Popular web testing framework
- Launches a browser → full Javascript support
- Selenium IDE for recording and running tests → quickly create tests
Selenium RC

- API for launching and controlling the browser
- Supports multiple programming language
- API:
  - click(), type()
  - waitForPageToLoad()
  - isVisible (), isPresent()
Selenium RC API + Selenium IDE = trouble

- API is very level:
  - Obscure tests
  - Lots of duplication
- You can quickly generate tests with Selenium IDE
- The result:
  - Large amounts of difficult to maintain test code
  - Very fragile tests ⇒ small change to UI, many broken tests
An example of bad test code

```java
public class ExampleOfBadWebTests extends AbstractSeleniumTest {
    @Test
    public void testCreateProject() {
        open("/ptrack/");
        type("j_username", "proj_mgr");
        type("j_password", "faces");
        clickAndWait("Login");
        assertTextPresent("(proj_mgr)");
        clickAndWait("link=Create New");
        String projectName = "XXX Project" + System.currentTimeMillis();
        type("projectDetails:nameInput", projectName);
        select("projectDetails:typeSelectOne", "label=External Desktop Application");
        ...
        clickAndWait("projectDetails:save");
        assertTextPresent("Inbox - approve or reject projects");
        assertTextEquals(projectName, "inboxPage:inboxTable:2:projectName");
        clickAndWait("inboxPage:inboxTable:2:details");
        assertTextPresent(projectName);
        assertTextPresent("External Desktop Application");
        assertTextEquals("Sean Sullivan", "detailsPage:initiatedBy");
        ....
        assertTitle("ProjectTrack - Project details");
        clickAndWait("detailsPage:ok");
        clickAndWait("link=Logout");
        assertTextPresent("Welcome to Project Track");
    }
}
```

Easy to write – record with Selenium IDE
But imagine coming back to this three months later ....
Improving tests with utility methods

- Write Test Utility methods = A Domain-Specific Language
  - Have intention revealing names
  - Call Selenium APIs

- Examples:
  - login()
  - goto...()
  - assertOn...Page()
  - logout()

- Write tests in terms of those methods
- Move those methods into a common superclass when appropriate
Improved example

```java
public class ImprovedExampleOfTests extends AbstractSeleniumTest {

    @Test
    public void testCreateProject() {
        login();

        createProject();
        assertProjectDisplayedInInbox();

        viewProjectDetails();
        assertProjectDetailsDisplayed();

        returnToInbox();
        logout();
    }
}
```

private void createProject() {
    clickAndWait("link=Create New");
    projectName = "XXX Project" + System.currentTimeMillis();
    type("projectDetails:nameInput", projectName);
    select("projectDetails:typeSelectOne", "label=External Desktop Application");
    type("projectDetails:requirementsInput", "Chris Richardson");
    type("projectDetails:requirementsEmailInput", "chris@chrisrichardson.net");
    ...
    clickAndWait("projectDetails:save");
}

Test is a lot more readable
Intention is clear
Less duplication
Less fragile
Intelligently evolve the language

- Write/record tests using low-level APIs
- Use Extract Method refactoring to create the utility methods
- Move methods into
  - A common superclass
  - Test Helper classes
Better ways to handle test data

- Web tests need data too
  - Filling in forms
  - Asserting the contents of the page
- Data embedded in code
  - Test data is sprinkled through application
  - Difficult to manage
  - Duplication
  - ...

Colorado Software Summit: October 19 – 24, 2008

© Copyright 2008, Chris Richardson
Example helper method

```java
public class ExampleOfWebTests extends AbstractSeleniumTest {

    private String projectName;

    public void createProject() {
        clickAndWait("link=Create New");
        projectName = "XXX Project" + System.currentTimeMillis();
        type("projectDetails:nameInput", projectName);
        select("projectDetails:typeSelectOne",
                "label=External Desktop Application");
        type("projectDetails:requirementsInput", "Chris Richardson");
        ...
        clickAndWait("projectDetails:save");
    }
}
```
Using domain objects in web tests

- Test utility methods:
  - Use domain objects created by mothers
  - Populate forms
  - Verify the contents of a page

- Benefits:
  - Improves readability
  - Improves management of test data
  - Parameterized methods are reusable
A much better example

public class ExampleOfGoodWebTests extends AbstractSeleniumTest {

    @Test
    public void testCreateProject() {
        login();
        Project projectToCreate = ProjectMother.makeNewProject();
        createProject(projectToCreate);
        assertProjectDisplayedInInbox(projectToCreate);
        viewProjectDetails(projectToCreate);
        assertProjectDetailsDisplayed(projectToCreate);
        returnToInbox();
        logout();
    }

    private void assertProjectDetailsDisplayed(Project projectToCreate) {
        assertTextPresent(projectToCreate.getName());
        assertTextPresent(projectToCreate.getType().getDescription());
        assertTextEquals(projectToCreate.getInitiatedBy().getFullName(),
                         "detailsPage:initiatedBy");
        ...
    }
}
Agenda

- Tests - a double-edged sword
- Taming test fixture logic
- Simplifying verification code
- Writing web tests
- **Testing Ajax applications**

Colorado Software Summit: October 19 – 24, 2008
The challenge of Ajax

- Ajax applications behave differently
- JavaScript executes after the page loads ⇒ less deterministic, predictable behavior
- Clicks don't result in page loads
  - Triggers an Ajax request that updates the same page
- DOM nodes are often hidden rather than non-existent
  - `assertElementPresent()` ⇒ true even when the element is not visible
Testing Ajax applications

- **Bad approach:**
  - Put lots of long sleeps in your code
  - Slows down the tests unnecessarily

- **Improved approach:**
  - Loop testing for element **visibility** with a short sleep
  - Use Selenium-RC "wait" feature
Messy Example

```java
protected void waitForVisibility(String selector) {
    WaitForElementVisible waiter = new WaitForElementVisible(selector);
    try {
        waiter.wait(String.format("Cannot find element <\%s>", selector), 3000);
    } catch (Wait.WaitTimedOutException e) {
        return;
    }
}

class WaitForElementVisible extends Wait {
    private final String selector;

    public WaitForElementVisible(String selector) {
        this.selector = selector;
    }

    @Override
    public boolean until() {
        return selenium.isElementPresent(selector) && selenium.isVisible(selector);
    }
}

class TestMothers {
    public void testSomething() {
        ...
        waitForVisibility("someForm");
        assertTrue(selenium.isVisible("someForm"));
        ...
    }
}
```

This works but the code quickly becomes cluttered with calls to `waitForXXX()`
Better: Implementation #2

- Encapsulate the waiting/loop within Test Utility methods, e.g.:
  - `assertElementVisible(...)`

- Benefits:
  - Simplifies the test code
A simple example

```java
protected void waitForVisibility(String selector) {
    WaitForElementVisible waiter = new WaitForElementVisible(selector);
    try {
        waiter.wait(String.format("Cannot find element <%s>", selector), 3000);
    } catch (Wait.WaitTimedOutException e) {
        return;
    }
}

class WaitForElementVisible extends Wait {
    ...
}

public void assertElementVisible(String selector) {
    waitForVisibility(selector);
    assertTrue(selenium.isVisible(selector));
}

public void testSomething() {
    ...
    assertElementVisible("someForm");
    ...
}
```

Simplifies the test code
Summary of web testing architecture

Tests

Application-specific Utility methods
e.g. login(), logout(), ...

Wrapped selenium APIs
e.g. hides Ajax related timing issues etc

Selenium RC
Some useful frameworks

- Umangite – Selenium web tests
  - code.google.com/p/umangite/
- ORMUnit – Persistence tests
  - code.google.com/p/ormunit
- Arid POJOs – Generic DAOs
  - code.google.com/p/aridpojos/
- And, others:
  - code.google.com/u/chris.e.richardson/
Summary

- Messy tests will kill your application
- Aggressively refactor tests to keep them simple
- Define classes with fluent interfaces
- Use Object Mothers to avoid duplication of test fixture logic
- Aggressively use Test Utility Methods:
  - Simplify web tests
  - Hide Ajax-related issues
For more information

- Buy my book 😊
  - Go to http://manning.com

- Send email:
  - chris@chrisrichardson.net

- Visit my website:
  - http://www.chrisrichardson.net

- Talk to me about consulting and training