Application-Fixture-Test: Building a Robust UI-Testing Architecture

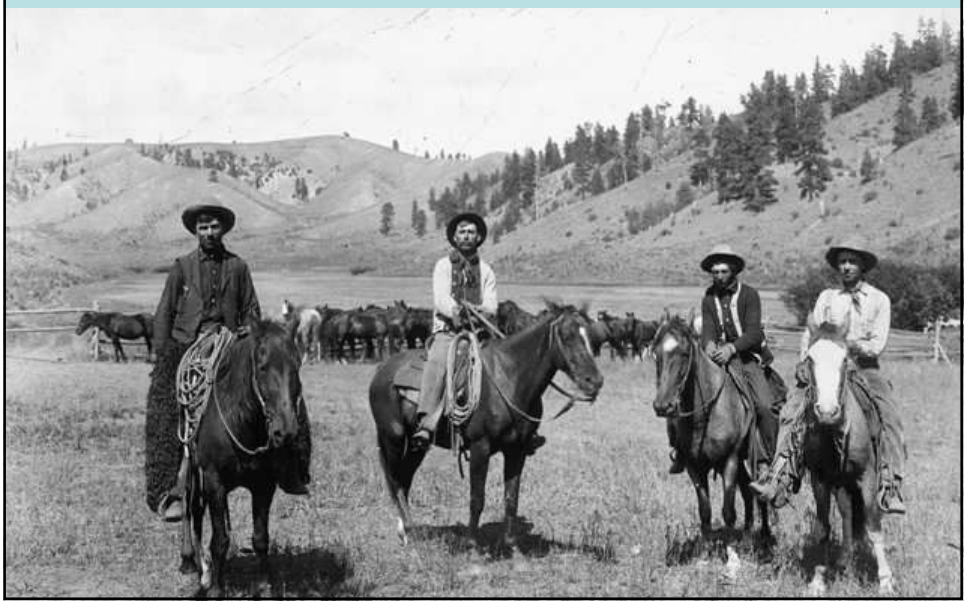
(Or How to Stop Worrying and Love Automated UI-Testing)

Phil Quitslund

Rob Ryan

{phil_quitslund, rob_ryan}@instantiations.com

UI Testing in the Wild



UI Testing in the Wild

UI Tests in practice are undisciplined, untrusted and burdensome.

Tests are generally hard to write, hard to understand, and hard to maintain.

In the worst cases, tests are delivering very little value.

Root Causes

Tests are often doing the ***wrong things*** (in the wrong ways).

Tests are often written by (and/or for) the ***wrong people***.

Proposed Solution

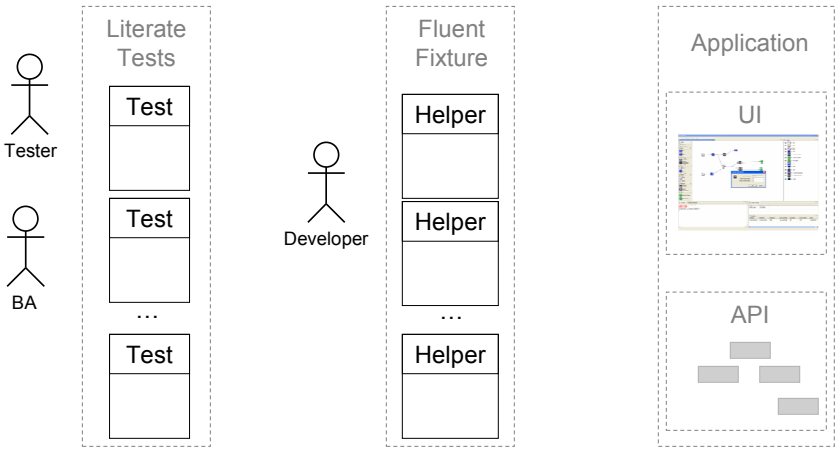
An architecture that partitions UI testing into two activities.

Writing Tests: Testers write tests in an application-specific test language.

Writing Fixtures: Developers build fixtures that make up the primitives in the test language.

Tests *and* fixtures are first class development artifacts.

Application/Fixture/Test



Example: Creating a Project (Free)

```
@Test
public void verifyProjectCreation() {
        click(menu("File/New/Other..."));
        waitFor(shellShowing("New"));
        click(tree("General/Project"));
        click(button("Next"));
        enter("MyProject");
        click(button("Finish"));
        waitFor(shellDisposed("New Project"));
        assertThat(projectExists("MyProject"));
}
```

Create Project Test (w/o Fixture)

Example: Creating a Project (Fixture)

```
import static WorkbenchHelper.*;

@Test
public void verifyProjectCreation() {
        createProject("MyProject");
}
```

VerifyProjectCreation.java

```
public static void createProject(String projectName) {
        click(menu("File/New/Other..."));
        waitFor(shellShowing("New"));
        click(tree("General/Project"));
        click(button("Next"));
        enter(projectName);
        click(button("Finish"));
        waitFor(shellDisposed("New Project"));
        assertThat(projectExists(projectName));
}
```

WorkbenchHelper.java

Payoff: Fixture Reuse

```
public void createProject(String projectName) { ...}
public void createFile(String filePath) { ...}
```

WorkbenchHelper.java

```
import static WorkbenchHelper.*;

@Test
public void verifyFileCreation() {
        createProject("MyProject");
        createFile("MyProject/myFile.txt");
}
```

VerifyFileCreation.java

Pattern: Intention-Revealing Fixture:

Fixture methods should have revealing names.

Payoff: Test Maintenance

```
public static void createProject(String projectName) {
        click(menu("File/New/Other..."));                  (1) → Menu change
        waitFor(shellShowing("New"));                      (2) → Shell name change
        click(tree("General/Project"));                    (3) → Tree path change
        click(button("Next"));                             (4) → Button text change
        enter(projectName);                                (5) → Focus change
        click(button("Finish"));                           (6) → Button text change
        waitFor(shellDisposed("New Project"));
        assertThat(projectExists(projectName));
}
```

WorkbenchHelper.java

The Rub:

Test infrastructure should not leak!

Payoff: Test Simplicity

```
import static WorkbenchHelper.*;

@Test
public void verifyFileCreation() {
        createProject("MyProject");
        createFile("MyProject/myFile.txt");
}
```

VerifyFileCreation.java

Pattern: Self-Verifying Fixture:

Fixtures enforce their own contracts.

Pattern: Externalized Configuration:

Fixtures encapsulate configuration details (locale, OS, app version, etc.)

A/F/T Process Summary

Testers write literate tests built out of primitives provided by the fixture.

Fixtures are delivered and maintained by developers.

Fixtures are a model, interface and contract.

Tests vs. Fixtures

Tests are

literate, declarative, descriptive (DAMP)

> A well designed Domain Specific Language will appear as **D**escriptive **A**nd **M**eaningful **P**hrases.
>
> http://blog.jayfields.com/2006/05/dry-code-damp-dsls.html

Fixtures are

intention-revealing, robust, DRY

> **Don't Repeat Yourself** (**DRY**, also known as **Once and Only Once** or **Single Point of Truth (SPOT)**)
>
> http://en.wikipedia.org/wiki/Don't_repeat_yourself

Challenge: Breaking Down the Wall

Requires an investment in change.

New kind of collaboration between QA and dev.

http://en.wikipedia.org/wiki/Image:Greatwall_large.jpg

Process Patterns

1. Fixture as Deliverable
2. Lockstep Delivery
3. Fixture Failure Escalation

Fixture as Deliverable

How do you ensure fixtures are well-factored?

Treat fixtures as deliverables.

(Estimate, schedule, review, etc.)

Lock-step Delivery

How do you ensure testers are never fixture-starved?

Bundle fixture deliveries with the associated functionality.

Fixture Failure Escalation

How do you ensure that fixtures stay in sync?

Run regular (full coverage) fixture smoke tests and treat failures as developer P1s.

Prime directive: protect your client (QA).

Key Points

Fixtures reify a model of the application under test

Fixture model should be defined in terms of domain expert's vocabulary (A DSL for testing)

Fixtures are the *only* way for testers to access the application

Fixtures are built and maintained by the same developers who deliver functionality

Fixtures are deliverables (need to be estimated, scheduled for, reviewed, etc.)

Tests might be DAMP but fixtures are DRY

- improves maintainability since logic that is most likely to change (and has the broadest impact) is not repeated