FINDING THE SCOPE

Patterns and best practices for testing Eclipse RCP Applications

Matthias Kempka (mkempka@eclipsesource.com)

Thursday, September 9, 2010
Why is testing RCP Applications a topic?
WHAT IS A SCOPE?
WHAT IS A SCOPE?

AND WHY IS IT IMPORTANT?
Advantage:
✓ Testing on method/class scope
✓ Easy to execute in IDE
✓ Fast
SCOPE: UNIT-TEST

Common problems:
- Code in legacy systems
- Class dependencies
- Misused as Mini-integration tests
UNIT-TESTS

Finding the scope of a unit test
UNIT TESTING WORKFLOW

Demo: A plain JUnit Test grows into a PDE Test
A COMMON MISUNDERSTANDING

I want to write a Unit Test

PDE Tests are the Eclipse way to write unit tests

PDE Tests allow execution of code in the product scope

I’ll just write an integration test
THE MISUNDERSTOOD PDE TEST

PDE Test

Product

Network

Database
THE MISUNDERSTOOD PDE TEST

This is no Unit Test!
UNIT TEST

• A unit is a small chunk of code
• Should be tested independent from other code
• That’s not always possible, especially in systems that where not created test driven
public static boolean needsLogin(IConfiguration config, ILocalProfile profile) {
    if (config.getForceLogin()) {
        return true;
    }
    if (profile == null) {
        return false;
    }
    if (!profile.getShared()) {
        return false;
    }
    if (profile.getType().equals(Open)) {
        return false;
    }
    return true;
}
An **Interception Point** is a point in your program where you can detect the effects of a particular change. (Michael Feathers, *Working Effectively with Legacy Code*)
public class Invoice {

    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                originator.getState().equals("NY")) {
                total.add(getLocalShipping());
            } else {
                total.add(getDefaultShipping());
            }
        } else {
            total.add(getSpanningShipping());
        }
        total.add(getTax());
        return total;
    }
}
INTERCEPTION POINT

• The method on the previous slide needs some changes.
• On the way it shall be refactored into this method:

```java
public class Invoice {
    public Money getValue() {
        Money total = itemsSum();
        total.add(shippingPricer.getPrice());
        total.add(getTax());
        return total;
    }
}
```
EFFECTS OF GETVALUE

• `getValue()` will change.

• The only use is in another class `BillingStatement`
EFFECTS ON GETVALUE

• Several other things will change that affect `getValue`.

```
getValue
```

```
shippingPricer
```

```
constructor
```

creates
A CHAIN OF EFFECTS

getValue

shippingPricer

creates

constructor

BillingStatement.makeStatement
INTERCEPTION POINTS

- getValue
- shippingPricer
- BillingStatement.makeStatement
- constructor

Thursday, September 9, 2010
INTERCEPTION POINTS

• Pick your interception point close to the change points

• **Safety**: Every step between a change point and an interception point is like a logical argument

• **Practicability**: In general (not always) it’s harder to set up interception points that are far away

• **Maintainability**: Your tests serve as regression tests. You don’t want to observe more effects than necessary.
A JUNIT FEATURE

JUnit Rules
JUNIT RULES

• Rules have been (unnoticed) in JUnit for a while (since 4.7)
• A simple way to get code run before and after the test
  • In the past, test runners were used for that
• A rule is a field, annotated with @Rule

• must be public

```java
@Rule
public TemporaryFolder tmpfolder = new TemporaryFolder();

@Test
public void useFiles() throws Exception {
    File propsfile = tmpfolder.newFile("myfile.properties");
    // do something with the file
}
```
PREDEFINED RULES

JUnit comes with a few rules predefined

- TemporaryFolder - Provides files that live as long as the test
- ExpectedException - A replacement for @Test(expected=...)
- TestName - Provides access to the test name
- Timeout - A replacement for @Test(timeout=...)
- ErrorCollector - Collect test failures instead of failing at the first error
CREATING A RULE

• Creating a rule is done by implementing `org.junit.rules.MethodRule`

• If you simply want to execute something before and after a test method, extend `org.junit.rules.ExternalResource`
A RULE FOR TESTS WITH SWT

```java
public class SWTShell extends ExternalResource {
    private Shell parent;

    @Override protected void before() throws Throwable {
        Shell shell = new Shell(Display.getCurrent());
        shell.pack();
        shell.setVisible(true);
        setParent(shell);
    }

    @Override protected void after() {
        getParent().dispose();
    }

    public Shell getParent() {
        return parent;
    }
}
```
INTERCEPTION POINTS

Demo: Testing View code without workbench
UNIT TESTS IN THE CONTINUOUS INTEGRATION

• If set up correctly, Unit tests generally can run as PDE tests

• Set up a test suite that runs all Unit and PDE tests as PDE tests

• Alternative: Put all the plug-ins on the java classpath and run the JUnit Tests in a normal Java environment (common solution in OSGi and RAP applications)
SCOPE: INTEGRATION TEST
Advantage:
✓ Regression tests on a scope where you don’t execute on a per-day basis
✓ High trust factor
PDE TEST AS INTEGRATION TEST

Product

Network

Database
PDE TEST AS INTEGRATION TEST

- PDE Test
- OSGi Container
- Product
- Network
- Database

✓ ✓

Thursday, September 9, 2010
PDE TEST AS INTEGRATION TEST

• Common problems:
  
  - A PDE Test can’t do much more than verify that a configuration exists
  
  - Setting external resources up (and cleaning them up) often must be done external
COMMON SOLUTIONS

• Common solutions for integration tests include

  ✓ Having a fixed test user/test data in the development database

  ✓ Scripts that can set up and tear down the environment

    - Hard to set up and maintain
SCOPE: FUNCTIONAL TEST
FUNCTIONAL TESTS

Advantage:
✓ Test on the same abstraction level as the user sees it
✓ High trust factor
SCOPE: FUNCTIONAL TESTS

Common problems:
- Which tool is the right one?
- Executing in IDE vs automated environment
FUNCTIONAL TEST TYPES

• Functional tests can be created through
  • Programming
  • Capture/Refactor/Replay

• In any case you need another tool than plain JUnit/PDE Test
SWTBot

- SWTBot finds SWT Widgets
- It provides an API for using widgets as if you were a user
- http://www.eclipse.org/swtbot/
Demo: SWTBot Test Case
ACCESS TO CODE

• The code for the Rule CaptureScreenshotOnFailure can be found at http://eclipsesource.com/blogs/2010/09/09/capture-screenshot-on-failing-swtbot-tests/
WHY NOT FUNCTIONAL TESTS?

• Good Unit-Tests can achieve >80% code coverage

• With a careful design, all controller and model logic can be tested

• You won’t be able to test Layouting with functional tests
WHY FUNCTIONAL TESTS

• Is the controller logic attached to the UI?
• Are various code units connected?
• Regression tests for tricky passages
PROPERTIES OF FUNCTIONAL TESTS

- Functional tests are several orders of magnitude slower than unit tests
- Immediate feedback almost impossible
- Interception points for functional tests are far away from the actual code
- Nothing (automated) is closer to the user experience
TEST PROJECT STRUCTURE

Test Setup in RCP Application
• We don’t want to ship JUnit with the application
• We want to use PDE tests and the Eclipse Testing Framework
TESTS IN SEPARATE BUNDLES

Bundle A

Test Bundle

JUnit

Bundle B
TESTS IN SEPARATE BUNDLES

• Advantages:

✓ A Test plug-in for a bunch of bundles

✓ Separate plug-in for Unit Tests

✓ Rather easy to set up and maintain plug-in structure
TESTS IN SEPARATE BUNDLES

• Consequences:
  - Hard to access internal classes (need to be exported)
  - Every method under test must be public
USES IN RCP APPLICATIONS

• Works well to some extend

• You’ll find a lot of code like this:

```java
/* public visibility for testing reasons */
public boolean isSaved() {
    return saved;
}
```
USES IN RCP APPLICATIONS

• Works well to some extend

• You’ll find a lot of code like this:

```java
/* public visibility for testing reasons */
public boolean isSaved() {
    return saved;
}
```

• Or worse, like this:

```java
/* public visibility for testing reasons */
public boolean saved;
```
USES IN RCP APPLICATIONS

• Your Manifest.mf contains a lot entries like this:

```text
Export-Package: org.eclipse.mail.client;
x-friends:="org.eclipse.mail.client.testplugin"
```
TESTS IN FRAGMENTS

Bundle A → Test Fragment A → JUnit

Bundle B → Test Fragment B → JUnit
TESTS IN FRAGMENTS

• Advantages:

✓ No classloader between test and class

→ We can narrow down the visibility to default

```java
/* default visibility for testing reasons */
boolean isSaved() {
    return saved;
}
```
TESTS IN FRAGMENTS

• Consequences:
  - Every bundle needs a separate test fragment
  - Creating and integrating bundles in the application becomes a heavy-weight task
  - Especially the initial setup frightens off developers
TESTS IN SEPARATE ARTIFACTS

- Bundle A
- Bundle B

Product

- Test Fragment A
- Test Fragment B

JUnit

Test Feature
CONTINUOUS INTEGRATION CONCERNS

• Executing Tests requires:
  • The RCP Application
  • The Test Feature
  • Eclipse Testing Framework
  • JDT + Requirements (This will likely vanish in 3.7)
CONTINUOUS INTEGRATION CONCERNS

• In practice, sometimes tests don’t get executed because of changed dependencies

• Hard to find out why

➡ Use p2 to install the tests into the product
TESTS IN THE PLUG-IN

Bundle A & Tests

Bundle B & Tests

JUnit

optional

optional
TESTS IN THE PLUG-IN

• Advantages:
  ✔ No test dependency management overhead
TESTS IN THE PLUG-IN

• Consequences:
  - Tests ship with the product
  - Hazzles with test/productive code interdependencies
USES IN OSGI APPLICATIONS

• This is a common structure for OSGi projects

• Not so common in RCP applications
• Don’t try to set up a Test Suite across different bundles yourself.

• There’ll be another talk about test suites later today

• Bundle Testcollector from Patrick Paulin makes it easy to set up Test Suites in an OSGi container
BUNDLE TESTCOLLECTOR

• Bundle Testcollector Input
  • pattern for bundle id
  • pattern for class name

• Goes through the specified bundles, pulls together the classes and puts them on a PDE test suite

• http://www.modumind.com/2008/06/12/running-unit-tests-for-rcp-and-osgi-applications/
BUNDLE TESTCOLLECTOR

• Small problem:
  • The Bundle Testcollector is only able to construct JUnit3 Test Suites
  • A small change is necessary to make it compatible with JUnit4 tests (wrap the found class in a JUnit4TestAdapter).
Demo: Setting up a Test Suite with BundleTestCollector
TEST SUITES

• Structure your test suites by execution speed and Test Runners

• SWTBot tests need a separate test runner

• Unit tests are meant to be fast, developers will execute them regularly

• Integration tests may take a while, they will mainly be executed in the continuous integration
ACCESS TO CODE

The Bundle Testcollector that was demonstrated can be accessed at http://eclipsesource.com/blogs/2010/09/09/an-almost-perfect-test-suite/
CONCLUSION

- Use the tools at hand
- Efficient testing comes with differentiation and structure
REFERENCES

• Michael C. Feathers, *Working Effectively with Legacy code*

• [http://eclipse.org/swtbot](http://eclipse.org/swtbot)