Testing Eclipse plug-ins: from unit to end-to-end testing

Lorenzo Bettini
DISIA – University Firenze, Italy

Vincenzo Caselli Francesco Guidieri
RCP-Vision, Firenze, Italy
Motivations

- Share some experiences/tips on testing Eclipse plug-ins:
  - Without an Eclipse instance when it’s not needed
  - Same for functional testing frameworks
  - Using plain JUnit tests as much as possible
  - When writing functional tests, only test our behavior
Main case study

- EMF Parsley https://www.eclipse.org/emf-parsley:
  - Quickly develop applications based on EMF models
  - Completely and easily customizable
  - Based on declarative customizations
  - Provides a DSL for easy configuration
  - Supports EMF persistences, XMI, CDO, etc.
  - Supports RAP

See also this afternoon talk

“The EMF Parsley DSL: an extensive use case of Xtext/Xbase powerful mechanisms”, 15:00 - 15:35, room “Argos”, XtextSummit
EMF Parsley

• Provides reusable and customizable Jface/SWT components
  - Tree
  - Form
  - Dialog
  - Editor
  - Combination of them
• Project wizard to get started
Many things to test...

- Mostly related to the UI...
  - But do we really need a running Eclipse while testing widgets’ behavior?
  - Do we really need functional tests?
    - Always?
In the beginning...

- We were testing almost everything with SWTBot
  - OK... but...
    - Took a lot of time
      - For writing a test case
      - For executing a test case
Improve the testing approach

- Split core and UI,
- We already did that as much as we could but...
- Most of Parsley important parts are based on the UI...
- Try to avoid SWTBot and programmatically check things with a Plug-in JUnit test?
  - Probably save some time when running tests...
    - It’s not SWTBot, it’s the time you need to start Eclipse during tests!
  - But even worse to write tests!
Example: create form controls

- We have a FormControlFactory that
  - Given an EMF object and one of its features
    - Create a form control based on feature
      - A checkbox for a boolean feature
      - A text for a string feature, etc.
    - Setup EMF databinding
      - If you change the control’s value, the EMF model is updated
      - And vice versa
You don’t need a running Eclipse!

- In order to test this scenario (including EMF databinding) you need
  - A Jface control
  - An EMF model
  - A Display
  - A (databinding) Realm
  - But not a running Eclipse!
public class DisplayHelperTestRule implements TestRule {

    private boolean displayOwner;
    private Display display;
    private Shell shell;

    public Display getDisplay() {
        if (display == null) {
            displayOwner = Display.getCurrent() == null;
            display = Display.getDefault();
        }
        return display;
    }

    public Shell getShell() {
        if (shell == null) {
            shell = createShell();
        }
        return shell;
    }

    @Override
    public Statement apply(final Statement base, Description description) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                try {
                    base.evaluate();
                } finally {
                    dispose();
                }
            }
        };
    }

    private Shell createShell() {
        // Create a new SWT shell
    }

    private void dispose() {
        // Dispose of SWT resources
    }

    // Other methods...
}

• Inspired by

http://www.codeaffine.com/2014/02/25/a-junit-rule-to-ease.swt-test-setup/
A JUnit Display rule

- Creates a Display and a Shell to be used as “parent” for SWT controls in your tests
- With some additional utility methods
  - e.g., to flush pending events in case of async ops

```java
public void flushPendingEvents() {
    while (Display.getCurrent() != null && !Display.getCurrent().isDisposed() && Display.getCurrent().readAndDispatch()) {
    }
}
```
And a Realm

- You need a databinding Realm for testing databinding
- See the Eclipse Wiki
  - https://wiki.eclipse.org/JFace_Data_Binding/Realm

```java
public class TestDefaultRealm extends Realm {
    private Realm previousRealm;

    public TestDefaultRealm() {
        previousRealm = super.setDefault(this);
    }

    @Override
    public boolean isCurrent() { return true; }

    @Override
    protected void syncExec(Runnable runnable) {
        runnable.run();
    }

    public void dispose() {
        if (getDefault() == this) {
            setDefault(previousRealm);
        }
    }
}
```
Some utility methods

- To run something in the UI thread

```java
/**
 * Executes the passed [@link RunnableWithResult} in a [@link Display#syncExec(Runnable)},
 * and returns the result; note that possible assertions within the runnable will NOT
 * make a test fail: the result will be null, and the exception will be logged.
 * 
 * @param toExecute
 * @return
 */

protected <T> T syncExec(final RunnableWithResult<T> toExecute) {
    final ArrayList<T> arrayList = new ArrayList<T>();
    getDisplay().syncExec(new Runnable() {
        @Override
        public void run() {
            try {
                arrayList.add(toExecute.run());
            } catch ( Throwable e) {
                LOGGER.error("Exception in runnable: "+ e.getMessage(), e);
                arrayList.add(null);
            }
        }
    });
    return arrayList.get(0);
}
```
protected void assertCheckbox(final Control control, final boolean checked) {
    assertControlClass(control, Button.class);
    final Button button = ((Button) control);
    syncExecVoid(new Runnable() {
        @Override
        public void run() {
            assertTrue("not a checkbox",
                (button.getStyle() & SWT.CHECK) != 0);
            assertEquals(checked, button.getSelection());
        }
    });
}

protected void assertLabel(final Control control, final String expectedText) {
    assertControlClass(control, Label.class);
    final Label label = ((Label) control);
    syncExecVoid(new Runnable() {
        @Override
        public void run() {
            assertEquals(expectedText, label.getText());
        }
    });
}
Now we can write JUnit tests

- We use **Xtend** ([https://www.eclipse.org/xtend](https://www.eclipse.org/xtend)) to write them
  - Java-like but More readable
  - Easier to write
  - Less verbose
  - Integrated with Java

- See also
  “Write cool scalable enterprise application tests with Xtend & embedded DSLs”, by Boris Brodski, EclipseCon Europe 2014 (PQD)
@Test def void testBooleanFeature() {
    // create a control for an EMF boolean feature
    val control = factory.createControl(testPackage.classForControls_BooleanFeature)
    // is it a checkbox unchecked?
    control.assertCheckbox(false)
    // change the EObject's value
    classForControlsInstance.booleanFeature = true
    // is the control updated?
    control.assertCheckbox(true)
}

@Test def void testBooleanFeatureReadOnly() {
    factory.readonly = true
    val control = factory.createControl(testPackage.classForControls_BooleanFeature)
    control.assertEnabled(false)
    control.assertCheckbox(false)
    classForControlsInstance.booleanFeature = true
    control.assertCheckbox(true)
}
Run it as a JUnit test

- For each Parsley core UI class
  - We have a JUnit test
  - That covers 100% that class
- Easy to write
- Amazingly fast to run!
- Just an excerpt
Example: create and test trees

- Given an EObject
  - We create a tree representation
    - With a content provider
    - With specific labels and images
      - With a label provider
    - When the model changes the tree must be updated
- How we test that?
  - Create a string representation of the tree
    - Represent children with indentation in the string
  - Compare the expected representation
  - Change the model and check that the tree is updated
Test the tree viewer

```java
@Test
def void testElementsAreRefreshedWhenNewElementIsAdded() {
    fillTestContainer
    val treeViewer = setupTreeViewer(testContainer.eResource, contentProvider)
    treeViewer.expandAll
    assertAllLabels(treeViewer,
        ...
        container
        elem1
        elem2
        ...
    )
    // tree contents are updated asynchronously
    execAndFlushPendingEvents[
        testContainer.classesForControls += createClassForControls
    ]
    assertAllLabels(treeViewer,
        ...
        container
        elem1
        elem2
        elem3
        ...
    )
}
```
Other similar tests

- JUnit tests for
  - Table viewers
  - Context menus
  - Etc.
Mock!

- We use mocking (Mockito)
  - For listeners
  - For events
    - Mouse events
    - Drag and Drop events
EMF Parsley testing framework

- We also release the testing framework we use
  - JUnit rules
  - Utility classes
  - Base classes
- Still under development
  - Use it at your own risk ;-)
- Feature to install:
  - “EMF Parsley Junit4 Support”
Testing a project builder

- You don’t need a functional testing framework
- But you need a JUnit Plug-in test of course

- Using Eclipse API:
  - Create projects programmatically
  - Create resources programmatically
  - Wait for the builder to finish building the workspace
  - Assert the possible error markers
Testing a project builder

- Use `.xtext.ui.testing` bundle, for example:
  - `org.eclipse.xtext.ui.util.PluginProjectFactory`
  - `org.eclipse.xtext.ui.testing.util.IresourcesSetupUtil.waitForBuild()`
  - `org.eclipse.xtext.ui.testing.util.JavaProjectSetupUtil.createJavaProject(String)`
  - `org.eclipse.xtext.ui.testing.util.PluginUtil.copyFileToWorkspace(Plugin, String, IProject, String)`
  - Etc.
def assertNoErrors() {
    val markers = ResourcesPlugin.getWorkspace().getRoot().findMarkers(IMarker.PROBLEM, true, IResource.DEPTH_INFINITE).
        filter[
            getAttribute(IMarker.SEVERITY, IMarker.SEVERITY_INFO) == IMarker.SEVERITY_ERROR
        ]
    assertEquals("unexpected errors:
        " +
        markers.map[getAttribute(IMarker.LOCATION) + ", " + getAttribute(IMarker.MESSAGE)].join("\n"),
        0,
        markers.size
    )
}
Testing a wizard

- You can test it without a functional testing framework:

```java
/**
 * Create the wizard dialog, open it and press Finish.
 */
def protected int createAndFinishWizardDialog(Wizard wizard) {
    val dialog = new WizardDialog(wizard.shell, wizard) {
        override open() {
            val thread = new Thread("Press Finish") {
                override run() {
                    // wait for the shell to become active
                    while (getShell() === null) {
                        Thread.sleep(1000)
                    }
                    getShell().getDisplay().asyncExec[
                        finishPressed();
                    ]
                }
            }
            thread.start();
            return super.open();
        }
    };
    return dialog.open();
}
...and use it like that
val wizard = ...create your wizard
wizard.init(PlatformUI.getWorkbench(),
    new StructuredSelection());
createAndFinishWizardDialog(wizard)
val project =
    ResourcesPlugin.getWorkspace().
   getRoot().getProject("TEST_PROJECT")
assertTrue(project.exists())
waitForBuild()
```
Functional tests

- In the end you might need functional testing frameworks
  - e.g., SWTBot, Jubula, RCPTT
- But test only your software behavior!
  - Let’s see some don’ts (in SWTBot)
  - See also

"Introduction to Functional Testing with SWTBot and Maven/Tycho" - EclipseCon Europe 2016

Some “don’t’s”

• You need to close the Welcome page (if present)
  – Don’t do that like that
    • It’s not something you want to test
    • You waste time due to timeouts if the Welcome page is not present

```java
@BeforeClass
public static void initBot() {
    bot = new SWTWorkbenchBot();
    try {
        bot.viewByTitle("Welcome").close();
    } catch (WidgetNotFoundException e) {
        // ignore
    }
}
```
Do it programmatically

```java
@BeforeClass
public static void initBot() {
    bot = new SWTWorkbenchBot();
    closeWelcomePage();
}

private static void closeWelcomePage() {
    for (SWTBotView view : bot.views()) {
        if (view.getTitle().equals("Welcome")) {
            view.close();
        }
    }
}
```
Some “dont’s”

- You need the Java perspective for testing your views, editors, etc.
  - Don’t open that simulating user interactions
  - That is not part of your functional tests!

```java
@BeforeClass
public static void initBot() throws InterruptedException {
    bot = new SWTWorkbenchBot();

    SWTBotMenu perspectiveMenu = bot.menu("Open Perspective");
    SWTBotMenu javaPerspectiveMenu = perspectiveMenu.menu("Java");
    if (javaPerspectiveMenu.isVisible() && javaPerspectiveMenu.isEnabled()) {
        javaPerspectiveMenu.click();
    }
}
```
private static void openJavaPerspective() throws InterruptedException {
    Display.getDefault().syncExec(new Runnable() {
        public void run() {
            try {
                IWorkbench workbench = PlatformUI.getWorkbench();
                workbench.showPerspective("org.eclipse.jdt.ui.JavaPerspective",
                        workbench.getActiveWorkbenchWindow());
            } catch (WorkbenchException e) {
                e.printStackTrace();
            }
        }
    });
}
By the way, some comparisons
Maven/Tycho

- For plain JUnit tests you may want to use maven-surefire-plugin instead of tycho-surefire-plugin
  - The latter still runs an Eclipse environment even without the UI
Conclusions

- Use plain JUnit tests as much as you can!
- Use Plug-in JUnit tests when you really need the workbench
  - Rely on existing utility functions for programmatically accessing workspace elements
- Use functional tests to test only your functional parts