Testing Eclipse plug-ins: Tips & Tricks

Lorenzo Bettini
DISIA – University Firenze, Italy

Vincenzo Caselli  Francesco Guidieri
RCP-Vision, Firenze, Italy

EclipseCon Europe 2017 – Project Quality Day
Motivations

• Share some experiences/tips on testing Eclipse plug-ins:
  – Without functional tests when they’re not needed
  – Without an Eclipse instance when it’s not needed
  – Using plain JUnit tests as much as possible
  – When writing functional tests, only test your behavior
Well-known best practice

- Separate Core logic from the UI
- Test the Core logic independently
- OK, and what about the UI?
  - Functional testing right away?
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?
SWTBot

- Test the UI with SWTBot
  - OK… but…
    - Takes a lot of time
      - For writing a test case
      - For executing a test case
Plug-in JUnit Tests

- Test the UI with a running Eclipse
  - OK... but...
    - Takes a lot of time
      - For writing a test case
      - For executing a test case
- The problem is not the functional testing framework...
Most of the UI parts only deal with Jface/SWT

In order to test these scenarios (including databinding) you need

- A Jface control
- Your domain 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();
        }
        ...  
    }

    @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() {
        return new Shell() {
            @Override
            public void open() {
                ...  
            }
        };
    }

    ...  

    private void dispose() {
        ...  
    }

    ...  

    // 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()) {
    }
}
```
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 (Exception 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(() -> {
        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(() -> {
        assertEquals(expectedText, label.getText());
    });
}
Plain JUnit

- Now you can run tests as plain JUnit tests!
- Fast fast fast!
Testing Databinding?

- You need a databinding Realm for testing databinding
- See the Eclipse Wiki
  - [https://wiki.eclipse.org/JFace_Data_Binding/Realm](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);
        }
    }
}
```
Testing Databinding

- Run in the UI and in the databinding Realm

```java
protected <T> T syncExecInRealm(final RunnableWithResult<T> toExecute) {
    final ArrayList<T> arrayList = new ArrayList<T>();
    getDisplay().syncExec(new Runnable() {
        @Override
        public void run() {
            Realm.runWithDefault(
                DisplayRealm.getRealm(Display.getDefault()),
                new Runnable() {
                    @Override
                    public void run() {
                        try {
                            arrayList.add(toExecute.run());
                            } catch (Exception e) {
                                LOGGER.error("Exception in runnable: "+ e.getMessage(), e);
                                arrayList.add(null);
                        } catch (Exception e) {
                            LOGGER.error("Exception in runnable: "+ e.getMessage(), e);
                            arrayList.add(null);
                        })
                        return arrayList.get(0);
                }
            });
        }
    }
```
Testing with Databinding

- Now create your controls inside the realm
  - `getShell()` is provided by the Display rule

```java
protected Control createControl() {
    return syncExecInRealm(() -> {
        // create your control to test, e.g.,
        return new Text(getShell(), SWT.BORDER);
    });
}
```
Example: test trees and tables

- Given a model
  - We create a tree viewer
    - 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
Example: test trees and tables

• Easy to create a string representation

```java
/**
 * A string representation of the table is built; this string representation is
 * then compared with the expected representation.
 *
 * @param tableViewer
 * @param expected
 */
protected void assertTable(TableViewer tableViewer, CharSequence expected) {
    assertEquals(expected.toString().trim(),
                tableItemsRepresentation(tableViewer.getTable().getItems()).trim());
}

/**
 * A string representation of the tree is built where children are indented of
 * TAB_INDENT number of tabs; this string representation is then compared with
 * the expected representation.
 *
 * @param treeViewer
 * @param expected
 */
protected void assertTree(TreeViewer treeViewer, CharSequence expected) {
    assertEquals(expected.toString().trim(),
                treeItemsRepresentation(getTreeItems(treeViewer)).trim());
}
```
@Test
public void testMyTree() {
    MyModel myModel = new MyModel();
    // fill the model with contents...

    TreeViewer treeViewer = new TreeViewer(getShell());
    // set label, content provider and input model...
    treeViewer.expandAll();

    assertTree(
        treeViewer,
        "my root
        " +
        "  my child1
        " +
        "  my child2"
    );

    // tree contents are updated asynchronously
    execAndFlushPendingEvents(() -> {
        // change the model, e.g., remove my child2
    });

    assertTree(
        treeViewer,
        "my root
        " +
        "  my child1"
    );
}
Other similar tests

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

- Use mocking (e.g., Mockito)
  - For listeners
  - For events
    - Mouse events
    - Drag and Drop events
Testing a project builder

- You don’t need a functional testing framework
- But in this case, you actually 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.

- Or get inspired by their implementations
@Inject
private PluginProjectFactory projectFactory;

public IProject createJavaPluginProject(final String projectName, final List<String> requiredBundles) {
this.projectFactory.setProjectName(projectName);
this.projectFactory.addFolders(newArrayList("src"));
// add your own project builder's id and nature
this.projectFactory.addBuilderIds(
    JavaCore.BUILDER_ID,
    "org.eclipse.pde.ManifestBuilder",
    "org.eclipse.pde.SchemaBuilder",
    YourOwnBuilder.BUILDER_ID);
this.projectFactory.addProjectNatures(
    JavaCore.NATURE_ID,
    "org.eclipse.pde.PluginNature",
    YourOwnNature.NATURE_ID);
this.projectFactory.addRequiredBundles(requiredBundles);
return this.projectFactory.createProject(new NullProgressMonitor(), null);
}
Asserting workspace errors

• Use Eclipse API to access error markers

```java
public void assertNoErrors() throws CoreException {
    List<IMarker> errorMarkers = Arrays.stream(IResourcesSetupUtil.root().
        findMarkers(IMarker.PROBLEM, true, IResource.DEPTH_INFINITE)).
    filter(
        marker ->
        marker.getAttribute(IMarker.SEVERITY, IMarker.SEVERITY_INFO)
        == IMarker.SEVERITY_ERROR).
    collect(Collectors.toList()));
    Assert.assertTrue(
        "unexpected errors: " + errorMarkers,
        errorMarkers.isEmpty());
}
```
Testing a wizard

- You can test it without a functional testing framework:

```java
/**
 * Create the wizard dialog, open it and press Finish.
 */
protected int createAndFinishWizardDialog(final Wizard wizard) {
    final WizardDialog dialog = new WizardDialog(wizard.getShell(), wizard) {
        @Override
        public int open() {
            final Thread thread = new Thread("Press Finish") {
                @Override
                public void run() {
                    while (getShell() == null) {
                        Thread.sleep(1000);
                    }
                    getShell().getDisplay().asyncExec(() -> finishPressed());
                }
            }
            thread.start();
            return super.open();
        }
    }
    return dialog.open();
    // ...and use it like that
    wizard.init(PlatformUI.getWorkbench(),
                new StructuredSelection());
    createAndFinishWizardDialog(wizard);
    project = ResourcesPlugin.getWorkspace().
                getRoot().getProject(TEST_PROJECT);
    assertTrue(project.exists());
    WaitForBuild();
    assertNoErrors();
```
Important

- These Plug-in JUnit tests must be run in the UI thread
  - Checkbox in the launch configuration
  - `<useUIThread>true</useUIThread>` in Tycho
Important

- While running SWTBot tests, they must NOT be run in the UI thread
  - Unselect the checkbox in the launch configuration
  - `<useUIThread>false</useUIThread>` in Tycho
PDE and Tycho

- If your projects are PDE plug-in projects
  - In Tycho they won’t work out of the box
    - The Eclipse target platform will be empty
    - You must set the target platform programmatically
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’t’s (in SWTBot)
  - See also

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

Some “dont’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
@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
- If the layout of Eclipse menus changes your tests will fail
  - That is not part of your functional tests!
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();
            }
        }
    });
}
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 (OSGI) environment even without the UI
Main case study

  - 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 last year’s talk

“How we reached 0 technical debt in our Eclipse project”, EclipseCon Europe 2016
EMF Parsley

- Provides reusable and customizable Jface/SWT components
  - Tree
  - Form
  - Dialog
  - Editor
  - Combination of them
- Project wizard to get started
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
Some comparisons

- SWTBot tests (left)
- JUnit tests (bottom)
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”
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
Evaluate the Sessions
Sign in and vote at eclipsecon.org
- 1  0  + 1