How we reached 0 technical debt in our Eclipse project

Lorenzo Bettini
Dip. DISIA, University of Firenze, Italy

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

EclipseCon Europe, 2016
Our Eclipse Project: EMF Parsley

- 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
EMF Parsley

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

- Delegates to \texttt{EMF.Edit} by default
- Customizations based on \textbf{Dependency Injection} (Google Guice), NOT on extension points
- \textbf{Declarative customizations} (polymorphic dispatch), NOT on instanceof cascades
- \textbf{Xtext/Xbase DSL}
  - Rich IDE tooling
  - Complete \textit{interoperability with Java}
  - Code Generation oriented (NOT reflective \Rightarrow \textbf{Debuggable}!)
Very quick demo

- For a wider presentation on Parsley, see tomorrow talk

“Six good reasons to spice up your projects with EMF Parsley”
Silchersaal - Thursday, October 27, 2016 - 13:00 to 13:35
Testing the DSL

- Xtext provides a nice framework for testing an Xtext DSL
  - With plain JUnit for the “compiler” part
  - With Plug-in JUnit tests for UI stuff like content-assist
- See my EclipseCon Europe 2015 talk
  - “Testing Xtext Languages”
  - Chapter 7, of my Xtext book, 2nd edition
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 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
    - Creates a form control based on feature
      - A checkbox for a boolean feature
      - A text for a string feature, etc.
    - Setups 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 EMF databinding you need
  – a Jface control
  – a model
  – a Display
  – a (databinding) Realm
A JUnit Display rule

- Creates a Display and a Shell to be used as “parent”
- 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()) {
    }
}
```

- Inspired by
And a Realm

- You need a databinding Realm for testing databinding
- See the Eclipse Wiki
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** to make them
  - More readable
  - Easier to write
  - Less verbose

- See also
  "Write cool scalable enterprise application tests with Xtend & embedded DSLs", by Boris Brodski, EclipseCon Europe 2014 (PQD)
Example

```scala
@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
  - Compare the expected representation
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
        
    )
}
```
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 ;-)
Functional Tests

- Then we have a bunch of SWTBot tests for the functional parts
  - Simulate what the users would do with our views/trees/dialogs, etc.
  - Simulate what the developers would do with a Parsley project
    - Create a new project with a wizard with a template
    - Import an example project set
    - Will everything compile in the workbench?
CDO Tests

• For most important tests
  – We have a version that uses CDO for EMF resources
  – With an in-memory database
Continuous Integration

- Maven/Tycho
- Hudson (Eclipse HIPP)
- Gerrit for code review
  - Hudson/Gerrit triggers
  - Nightly builds
    - All Tests in several Eclipse versions (Indigo, Kepler, ..., Neon)
  - Jacoco
  - Findbugs
  - Sonarqube
From testing to code quality

- Tests gave us a safety net for
  - Experimenting with new features
  - Refactoring to reach code quality standards
Sonarqube results

- ~95% code coverage
- 0 code duplication
- 0 dependency cycles
- ...
- 0 Technical Debt
## Sonarqube results

<table>
<thead>
<tr>
<th>Technical Debt Pyramid</th>
<th>Technical Debt</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Portability</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Maintainability</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Security</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Usability</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Efficiency</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Changeability</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Reliability</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Testability</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Sonarqube results

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory Tangle Index</td>
<td>0.0%</td>
<td>Cycles: 0, &gt; 0</td>
</tr>
<tr>
<td>Dependencies To Cut</td>
<td></td>
<td>Between Directories: 0, Between Files: 0</td>
</tr>
<tr>
<td>Duplications</td>
<td>0.0%</td>
<td>Lines: 0, Blocks: 0, Files: 0</td>
</tr>
</tbody>
</table>

### Technical Debt

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issues</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocker</td>
<td>0</td>
</tr>
<tr>
<td>Critical</td>
<td>0</td>
</tr>
<tr>
<td>Major</td>
<td>0</td>
</tr>
<tr>
<td>Minor</td>
<td>0</td>
</tr>
<tr>
<td>Info</td>
<td>0</td>
</tr>
</tbody>
</table>
Took some time to get there...

### Issues Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Issues</td>
<td>1,252</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Blocker issues</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Critical issues</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Major issues</td>
<td>1,075</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Minor issues</td>
<td>162</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Duplicated Lines Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicated lines (%)</td>
<td>7.1%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Duplicated lines</td>
<td>1,097</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Duplicated blocks</td>
<td>987</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Duplicated files</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
We’re currently not testing the RAP version of Parsley
  - We’re working on using Selenium for that
  - But it does not work out of the box
  - Manual tests show that RAP single sourcing works great!
Links

Homepage
www.eclipse.org/emf-parsley

Documentation
https://www.eclipse.org/emf-parsley/documentation.html

Forum

Bugzilla
https://bugs.eclipse.org/bugs/buglist.cgi?product=EMF.Parsley

Thanks!