Literate Testing

Peter Arrenbrecht, codewise.ch
peter.arrenbrecht@gmail.com

We all design APIs

Some of us teach them, too

Motivation

AFC has a set of default conventions for the layout of a spreadsheet which, when follo
ensure a certain consistency, and simplify the association of cells to inputs and outpu
users (they do not have to use cell names). It will be much easier for them to get started with these conventions if you provide them with ready-made templates.

It also greatly helps your users if they can start with a spreadsheet file that implements the computation the system is currently configured to perform. (Of course, this is only possible if this computation can be expressed in terms of a spreadsheet.)

Now, while you could certainly create these initial files by hand in Excel and ship them with your application, AFC supports generating them at run-time. This has the following advantages:

- The initial file can be generated in any of the spreadsheet file formats supported by AFC, as desired by the user.
- If the current computation is already customizable (see using AFC without a spreadsheet file), then generating the initial file for this computation must be done at run-time.

Generating The Internal Model

AFC generates initial files from its internal spreadsheet model. So in order to generate one, we first need to build its model in memory. This is exactly the same process as is needed when using AFC without a spreadsheet file. See there for details.

Generating The File

Once you have the internal model set up, you can tell AFC to write out a spreadsheet file for it. There are two flavors of this API. The simpler version automatically deduces the the spreadsheet file type by the file name extension (.xls, .xsd, etc.), and always writes to a file:

```
Spreadsheet s = buildSpreadsheet();
SpreadsheetCompiler.saveSpreadsheet( s, getOutputFile(), null );
```

The other version lets you specify the extension explicitly and returns the generated spreadsheet as a stream:

```
Spreadsheet s = buildSpreadsheet();
ByteArrayOutputStream os = new ByteArrayOutputStream();

SpreadsheetSaver.Config cfg = new SpreadsheetSaver.Config();
cfg.spreadsheet = s;
cfg.typeExtension = getSpreadsheetExtension(); // .xls or .ods
cfg.outputStream = os;
SpreadsheetCompiler.newSpreadsheetSaver( cfg ).save();
```

Refines

```
public interface SpreadsheetSaver
{

  /**
   * Configuration data for new instances of
   * {@link org.formulacompiler.spreadsheet.SpreadsheetSaver}.
   *
   * @author peo
   *
   * @see SpreadsheetCompiler#newSpreadsheetSaver(org.formu
   */
  public static class Config
  {

    /**
     * Mandatory internal spreadsheet model that should be writte
     * Normally constructed using a {@link org.formulacompiler.s
     */
    public Spreadsheet spreadsheet;
```

And we all test them

```
@Test
public void testGenerateFile() throws Exception
{
  // ---- GenerateFile
  Spreadsheet s = buildSpreadsheet();
  SpreadsheetCompiler./**/saveSpreadsheet/**/( s, getOutputFile(), null );
  // ---- GenerateFile
  SpreadsheetAssert.assertEqualSpreadsheets( s, new BufferedInputStream( ne
}
```

```
@Test
public void testGenerateStream() throws Exception
{
  // ---- GenerateStream
  Spreadsheet s = buildSpreadsheet();
  ByteArrayOutputStream /**/os/**/ = new ByteArrayOutputStream();

  SpreadsheetSaver.Config cfg = new SpreadsheetSaver.Config();
  cfg.spreadsheet = s;
  cfg./**/typeExtension/**/ = getSpreadsheetExtension(); // .xls or .ods
  cfg./**/outputStream/**/ = os;
  /**/SpreadsheetCompiler.newSpreadsheetSaver( cfg ).save();/**/
  // ---- GenerateStream
  SpreadsheetAssert.assertEqualSpreadsheets( s, new ByteArrayInputStream(
}
```

Verifies

Tutorials and Examples

This talk is about

- writing tutorials,
- with lots of tested examples
- and how this improves the quality of your APIs

Writing Tutorials Buys You

- Focus on users and their needs
  - Use cases, stories
  - Descend from on high
- Broadens view
  - Interplay of components
  - Entire use cases
- Invites cooperation
  - Reviewers, potential users
- Discourages bells and whistles
  - They all have to be explained

Giving Examples Buys You

- Consistent and convincing terminology
- Concise user code
  - Need a simplified API atop a lower-level one?
- Usage as intended
  - Correct and optimal
  - Similar code easier to maintain
  - Might be easier to refactor later

Testing Examples Buys You

- Completeness
  - No hand waving, no shortcuts
- Maintainability
  - Refactor text around examples too
  - Restarts the cycle of reflection induced by focus on tutorials
- Side benefits (not for your API as such)
  - No broken examples ever
  - Important set of tests

Real-Life Experience

- Excel to JVM Compiler (http://formulacompiler.org/)
- Extensive tutorial online
  - Led to layered API
  - Guides users towards best practices
  - Forced thoroughness on thorny features
- Giving talks about it
  - Identified need for more guidance
  - Meaning more building blocks

Code-First Techniques

- Code comments
  - Write use-case oriented tests
  - Document liberally
  - IDE support
- Formattable code comments
  - Produce external docs like JavaDoc
  - But for tutorials, not reference
  - Bumblebee
    - http://agical.com/bumblebee/bumblebee_doc.html

Prose-First Techniques

- Embedding into external documents
  - „Literate Programming“
  - Often no IDE support
  - Flow might be dictated by code
- Citing into external documents
  - IDE support for code, editor support for prose
  - Citations may not be visible while writing
  - JCite
    - http://arrenbrecht.ch/jcite/

Real-Life Experience

- Documented JCite's own extensibility
- First in code-first fashion; found
  - Bad terminology
  - Bad abstractions forcing user to duplicate code
- Then in prose-first fashion; found
  - More bad terminology
  - Even some introduced just before

Demo

- Citing and highlighting
- Tripwires
- Ant integration

Considerations

- Code first
  - Simple and quick to set up
  - Tutorials available directly in IDE
- Prose first
  - Clean slate for teaching
  - Focus on story, background and goals
  - Names are better when created while writing story
  - Condense examples, exclude clutter
  - Produces polished presentation
  - Encourages rereading and refinement

Code-First Recommendations

- Write tests first
- Write test introduction first (clean slate)
- Hide clutter (e.g. in base classes)
- Separate tutorials from technical tests
- Don't misuse JavaDoc for lengthy tutorials
  - Refer to tutorial tests instead

Prose-First Recommendations

- Start at high level (system overview, goals)
- Sketch examples directly in text editor at first
  - Remain focused on user story
- Cite test assertions
  - Does this assertion really test what I'm claiming?
- Cite as much as possible (no copy/paste)
- Return to high level
  - How did we address our goals? Link to examples.
- Include in source repository

Caveat Emptor

- Short samples vs. sound API design
  - Don't make sample code short at all costs
  - Be very wary when adding a convenience API
- Tutorials don't obviate need for doc comments
- Not every programmer is good at prose
  - Use reviews, have editors
  - Give talks, in-house presentations
  - But don't evade the thinking involved in teaching

Coming Full Circle

(What this talk taught me)

- How can we integrate external tutorials with IDEs?
- I should mandate tutorial-style tests when reviewing library code

Links

- JCite
  - Cite fragments into HTML documents
  - http://arrenbrecht.ch/jcite/
- Bumblebee
  - Produce HTML from code comments
  - http://agical.com/bumblebee/bumblebee_doc.html
- Literate Testing
  - http://arrenbrecht.ch/testing/

Thank you!