Literate Testing

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

Einige von uns erklären sie auch

Motivation

AFC has a set of default conventions for the layout of a spreadsheet which, ensure a certain consistency, and simplify the association of cells to inputs 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();
```

Verbessert

Wir alle designen APIs

```
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;
```

Prüft

Und wir testen sie

```
@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(
}
```

Erklärungen und Beispiele

In diesem Vortrag geht es um das

- Erklären beim Schreiben von Anleitungen, das
- Zeigen anhand von Beispielen und das
- Prüfen der Beispiele in automatisierten Tests.

Aber eigentlich geht es darum, wie dies die Qualität von APIs verbessert.

Wie hilft das Erklären dem API?

- Fokus auf Anwender und ihre Bedürfnisse
  - *Use Cases, User Stories*
  - Dann, wenn man noch Zeit hat
- Erweitert die Perspektive
  - Zusammenspiel von Komponenten
  - Gesamtabläufe
- Immer wieder!
  - Mildert Gefahr, zu früh in Implementationsdetails abzutauchen

Erklären, die Zweite

- Fördert Zusammenarbeit
  - Vom Thron steigen
  - Reviews der Usability des APIs
- Macht ehrlich und überzeugend
  - Man muss für seine Arbeit einstehen
  - Speziell in Vorträgen!
- Bremst Featuritis
  - Das muss nun auch noch alles erklärt werden

Was bringen die Beispiele?

- Konsistente und überzeugende Begriffe
  - Beispielcode erzählt Teil der Geschichte
  - kryptische Hacks stechen raus
- Prägnanter Anwender-Code
  - Ein vereinfachtes API für typische Fälle?
- im Sinne des Erfinders
  - Korrekte und effiziente Verwendung
  - Gleichartiger Code einfacher zu Warten
  - Kann späteres Refactoring vereinfachen

Und warum die Beispiele testen?

- Vollständigkeit
  - Kein Händewedeln, keine Abkürzungen
- Wartbarkeit
  - Beim Refactoring sieht man, wo Text betroffen ist
  - Führt zurück zur Reflexion anhand des Erklärens

- Seiteneffekte (helfen nicht dem API an sich)
  - Die Beispiele funktionieren. Immer.
  - Wichtiger Satz von automatischen Tests

Aus dem Leben gegriffen

- Excel zu JVM Compiler (http://formulacompiler.org/)
- Umfassende Anleitung im Netz
  - Führte zu vereinfachten APIs für typische Fälle
  - Lenkt Anwender zu bewährten Mustern
  - Zwang zur Gründlichkeit bei komplexen Features
- Erklären in Vorträgen
  - Zeigte Bedarf an weiterer Lenkung auf
  - D.h. es braucht weitere fertige Muster und Bausteine

Code-First Techniken

- Kommentare im Code
  - Schreibe am Use-Case orientierte Tests
  - Erkläre sie grosszügig in Kommentaren
  - Wird gut von der IDE unterstützt
- Formatierbare Kommentare im Code
  - Erzeuge externe Dokumente wie JavaDoc
  - Aber für Anleitungen, nicht Referenzen
  - Bumblebee
    - http://agical.com/bumblebee/bumblebee_doc.html

Prose-First Techniken

- Beispiele in Dokumente reinschreiben
  - „Literate Programming“
  - Selten von IDE unterstützt
  - Textaufbau kann von Code-Struktur bestimmt sein
- Beispiele zitieren
  - IDE für den Code, Dokumenteneditor für den Text
  - Zitate evtl. nicht direkt sichtbar beim Schreiben
  - JCite
    - http://arrenbrecht.ch/jcite/

Ein Experiment dazu

- Dokumentierte die Erweiterbarkeit von JCite
- Zuerst mit Code-First fand ich
  - Schlechte Begriffe
  - Schlechte Abstraktionen, die zu Redundanz im Anwender-Code führten
- Dann mit Prose-First fand ich
  - Noch mehr schlechte Begriffe
  - Sogar solche, die ich mit Code-First gerade eben eingeführt hatte!

Demo von JCite

- Zitieren und Hervorheben
- *Tripwires* - wo muss man Text prüfen?
- Integration mit Ant

Vergleich der Techniken

- Code-First
  - Einfach und schnell eingerichtet
  - Anleitungen sind direkt in der IDE verfügbar
- Prose-First
  - Leeres Blatt führt zu tieferer Reflexion
  - Fokus auf *User Story*, Hintergrund und Zielen
  - Begriffe besser, wenn im Schreibfluss gewählt
  - Verknappte Beispiele ohne Ballast
  - Ergibt ausgestaltete Dokumentation
  - Fördert erneutes Durchlesen und Verbessern

Empfehlungen zu Code-First

- Tests vor dem API schreiben
- Einführungstext zum Test noch vorher schreiben (leeres Blatt!)
- Ballast verbergen (z.B. in Basisklassen)
- Beispieltests von technischen Tests trennen
- JavaDoc nicht für Anleitungen verwenden
  - Stattdessen auf Beispieltests verweisen

Empfehlungen zu Prose-First

- Weit oben anfangen (Übersicht, Ziele)
- Beispiele zunächst direkt im Text skizzieren
  - So bleibt man auf die Story fokussiert
- Die Checks in den Tests auch zitieren
  - Prüft dieser Check wirklich, was ich behaupte?
- So viel wie möglich zitieren (kein Copy/Paste)
- Nach weit oben zurückkehren
  - Sind die Ziele erreicht? Verweise zu Beispielen.
- Mit dem Code einchecken

Caveat Emptor

- Kurze Beispiele auf Kosten des API
  - Es gibt noch andere Kriterien für ein gutes API
  - Vorsicht beim Einführen von vereinfachten APIs
- Anleitung ersetzt die Referenz (JavaDoc) nicht
- Nicht alle Programmierer schreiben gut
  - Mündlich: Vorträge halten, Präsentationen im Haus
  - Reviews, Lektoren aus Administration?
  - Der Reflexion nicht ausweichen!

Den Kreis schliessen

(Was hat mich dieser Vortrag gelehrt?)

- Wie können wir externe Anleitungen in IDEs integrieren?
- Wie die Zitate direkt im Dokumenteditor anzeigen?
- Wenn ich Library-Code reviewe, dann bestehe ich auf Beispieltests.

Links

- JCite
  - Zitiert Code-Ausschnitte in HTML-Dokumente
  - http://arrenbrecht.ch/jcite/
- Bumblebee
  - Erzeugt HTML aus Kommentaren im Code
    http://agical.com/bumblebee/bumblebee_doc.html
- Literate Testing
  - http://arrenbrecht.ch/testing/

Vielen Dank!